REAL-TIME 3D GRAPHICS FOR THE ATARI ST

a practical guide to 68000 assembler programming

Andrew Tyler

SIGMA PRESS

REAL-TIME 3D GRAPHICS FOR THE ATARI ST

– a practical guide to 68000 assembler programming

Andrew Tyler

SIGMA PRESS – Wilmslow, United Kingdom

Copyright ©, Andrew Tyler, 1991

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without prior written permission.

First published in 1991 by

Sigma Press, 1 South Oak Lane, Wilmslow, Cheshire SK9 6AR, England.

Reprinted 1992

British Library Cataloguing in Publication Data

A CIP catalogue record for this book is available from the British Library.

ISBN: 1-85058-217-3

Typesetting and design by

Sigma Hi-Tech Services Ltd

Printed in Malta by
Interprint Ltd.

Distributed by

John Wiley & Sons Ltd., Baffins Lane, Chichester, West Sussex, England.

Acknowledgement of copyright names

Within this book, various proprietary trade names and names protected by copyright are mentioned for descriptive purposes. Full acknowledgment is hereby made of all such protection.

Cover Picture

We are grateful to Database Interactive Ltd for their help in producing the cover photograph, which is based on an illustration from the "Flying Around the World" program described in this book.

Disclaimer

Whilst every effort has been made to ensure that the programs in this book work as described, it is not possible to test them under all possible conditions. Therefore, the programs are provided "as is" without warranty of any kind either express or implied. Neither the author nor the publisher assume any responsibility for their use nor for any infringements of patents or other rights of third parties which would result. Likewise neither the author nor the publisher accept responsibility for the accuracy of the information contained in this book.

Preface

"A picture is worth a thousand words". This statement sums it all up.

A few years ago, when I first opened a book on computer graphics, I was stunned by the beautiful simulations of life-like objects generated by computers. But these were from state-of-the-art machines, far more powerful than the popular personal microcomputers of the time, which were almost exclusively 8-bit.

With the advent of 16-bit micros things changed markedly. Their extra power and memory had an immediate impact on all graphics applications, from painting programs to fast flight simulators sporting solid 3D primitives (objects). The low price and high power of micros such as the Atari ST and the Commodore Amiga meant that anyone could enjoy high quality computer graphics (especially in games) for a few hundred pounds. But enjoying other peoples programs is only half the fun. Surprisingly, writing them is not really as difficult as it looks. Of course there is a fair amount of technology to be learnt along the way, but a good deal of the dramatic effect comes from the speed of the machines themselves, performing fairly standard algorithms very fast.

When I first became interested in graphics programming and wanted it to be as fast as possible in machine code, it seemed to me that essential information was spread thinly in the literature. There were certainly books on machine code programming and on computer graphics; there were even a few books on machine code graphics programming. But somehow I could never quite find the balance I was looking for. Standard texts on computer graphics seemed amazingly obscure on certain aspects of transforms, in particular how to picture a scene from an arbitrary view point. I felt, quite unreasonably perhaps, that there was a tendency to hide it all behind a smokescreen of professional mystique; certainly it helped considerably to understand the mathematics of vectors and matrices, but surely all this had been worked out years ago and ought to be fairly straightforward? Perhaps it was just

me! Anyway I wanted to write 3D solid graphics programs that would run in real time (like a flight simulator), and couldn't find anyone who would tell me how to do it. For sure the people who write commercial games knew, but they weren't telling - for obvious reasons! There were a few very useful serialised articles in magazines but, by necessity I'm sure, these were often too brief and not exactly what I wanted.

Things came to a head when I was assigned to give a college course on Advanced Microcomputer Software (which was another way of saying "Assembly Language Programming on the 68000"). Teaching programming, especially in assembly language, can be a very sterile pastime unless the application is interesting. What better application than graphics? and what better machine (for the price) than the Atari ST?

This book arose from my efforts to penetrate the world of computer graphics and make some of the basics understandable (I hope) to non-specialists. It is about fast 3D (so-called vector) graphics in assembly language. There is certainly no guarantee that the programs in this book are the most efficient, most elegant and fastest of their kind. But they are reasonably fast. Certainly as fast as some commercial programs! The astute reader will undoubtedly be able to make improvements (and tell me, I hope).

There is no assumption that the reader has any prior knowledge of any of the following subjects, all of which eventually figure heavily in the graphics process: the ST operating system, vectors and matrices. It helps enormously to have some knowledge of them, but those aspects which are important for the programming are explained in the text when they are used. There are further explanations in the Appendices. That is not to say that the book contains exhaustive discussions of these subjects, only sufficient for the purpose in hand. The enthusiast will undoubtedly wish to add to them.

As regards the assembly language, although an Appendix contains a list of the instruction set and (most important) the addressing modes, it is assumed that the reader who wishes to fully understand what is going on will have on hand a 68000 code reference book (they are available in pocket form very cheaply).

For the writing, assembly, debugging and running of the programs in the book the powerful and friendly Devpac ST 2 assembler from Hisoft has been used. This comes as an integrated package within which all functions can be performed. There are several good assemblers/debuggers available but I like this one best. It's an excellent workhorse for the development of assembly language programs. A demonstration version of it was provided on Cover Disk 10 of *ST Format*. A full working version of DevpacST 1, the first version of the assembler, but without the debugger, has been made available on the cover disk of the March 1991 issue of *Atari ST User*. It can be used to assemble the programs in this book, providing the headings SECTION TEXT, SECTION DATA and SECTION BSS are deleted

from them (or marked with an asterisk, *). Further information on the assembler is given in Appendix 2.

The book is laid out in serial form. Each chapter deals with a different topic and illustrates its application with example programs. To the experienced reader the early chapters will seem pedestrian. To the newcomer they will not. There is really no easy introduction to the overall process and so each stage (a somewhat artificial division) is dealt with in detail separately. Each stage of the graphics "pipeline" does a specific task and has its own algorithm and strategy. The chapters are laid out to reflect the build up of the overall process. Each chapter has its own example programs and the programs saved from the earlier chapters are used in later ones so that they don't have to be entered more than once. In this way the example programs at the end of the book end up being the largest and most complex, though the amount of code you have to enter for each new chapter doesn't really increase very much. The programs are written for the Atari ST but can be modified to run on any 68000 based computer since, with the exception of certain specifics concerned with the screen and operating system, the graphics routines are entirely independent and self-contained.

Computer graphics is a vast subject; a book of this length can only cover a small part. Especially since it is not just descriptive but contains working programs. Techniques such as Ray Tracing and Radiosity methods are perhaps better suited to a future, more powerful generation of personal computers. But that will come; it is likely that many of the software routines discussed here will be replaced in future machines by hardware "geometry engines".

Until then, 3D graphics will have to be done by "bashing the bytes". Good luck.

Andrew Tyler

THE DISK

The programs listed in this book are available on disk. The disk contains all the source files listed here. The disk can be obtained from:

LIVE GRAPHICS
PO BOX 19
ALDERLEY EDGE
CHESHIRE SK9 7XY

The price (at April 1992) is £6.99 inclusive in the UK.

CONTENTS

1. An Overview 1
1.1 A New Medium 2
1.1.1 Is it Art, or What? 2
1.2 What Can You Do With A 16-bit Micro? 5
1.3 Assembled for Speed 6
1.4 Writing for a 16 bit Micro 7
1.5 The Programs 8
1.6 The ST Operating System 8

2. Drawing on the Screen 10
2.1 The Screen 10
2.1.1 High Resolution 11
2.1.2 Low Resolution 12
2.1.3 Medium Resolution 13
2.2 The Line A routines 13
2.3 Example Programs 14
2.3.1 put_pixl.s 14
2.3.2 set_pixl.s 15
2.3.3 systm_00.s 15
2.3.4 data_00.s 16
2.3.5 polyfil0.s 16
2.3.6 polyfil1.s 16

3. Modelling a 3D World 25
3.1 3-D Modelling 26
3.2 Transformations and Frames of Reference 27
3.2.1 The Object Frame 27
3.2.2 The World frame 28
3.2.3 The View Frame 29
3.2.4 The Screen 29
3.3 Coordinate Systems 30
3.4 Vectors and Matrices 31
3.4.1 Vectors 32
3.5 Data Structures 32
3.5.1 Variables and Labels 32
3.5.2 Lists 33
3.6 Summary 35

4. Fast Filling a Polygon 36
4.1 Bresenham Algorithm for Drawing Lines 38
4.2 Tailoring Bresenham to the Polygon Fill 40
4.3 Example Programs 41
4.3.1 polyfil2.s 41
4.3.2 core_00.s 42
4.3.3 bss_00.s 43
4.3.4 systm_01.s 43

5. Windowing 51
5.1 Sutherland-Hodgman Clipping Algorithm 52
5.2 Example Program 55
5.2.1 clipfrme.s 55
5.2.2 core_01.s 56
5.2.3 bss_01.s 56

6. Getting Things Into Perspective 65
6.1 The Perspective Transform 65
6.2 Homogeneous Coordinates 68
6.3 Example program 70
6.3.1 perspect.s 71
6.3.2 data_01.s 71
6.3.3 data_02.s 72
6.3.4 bss_02.s 73
6.3.5 core_02.s 73

7. Simple Rotations 81
7.1 Geometric Transforms 81
7.2 Rotations About the Principal Axes 82
7.2.1 Rotation about the x-axis 83
7.2.2 Rotation about the y-axis 83
7.2.3 Rotation about the z axis 83
7.2.4 Composite Rotations 84
7.3 The Object-to-World Transform 85
7.4 Example Program 87
7.4.1 otranw.s 87
7.4.2 data_03.s 88
7.4.3 core_03.s 89
7.4.4 systm_02.s 89
7.4.5 bss_03.s 89

8. Keyboard, Joystick and Mouse 100
8.1 "Quick and Dirty" 100
8.2 Strictly by the Book 102
8.2.1 The Keyboard 102
8.2.2 The Joystick 102
8.2.3 The Mouse 103
8.3 Talking to the IKBD 103
8.4 Example Programs 104
8.4.1 ramview.s 104

8.4.2 key_peek.s 104
8.4.3 systm_03.s 104
8.4.4 joy_test.s 104

9. Hidden Surfaces and Illumination 110
9.1 Hidden Surface Removal. 111
9.2 Calculating the Surface Normal Unit Vector 113
9.3 Illumination and Colour 115
9.3.1 The Colour Palette 116
9.4 Example Programs 118
9.4.1 ill_hide.s 118
9.4.1 core_04.s 118
9.4.3 data_04.s 119
9.4.4 bss_04.s. 120

10. General Transforms in 3D 129
10.1 Geometric Transforms. 129
10.1.1 Rotations 130
10.1.2 Scaling. 133
10.1.3 Shear. 133
10.2 Instance Transforms 136
10.3 Physical Realism 137
10.4 Example Program. 137
10.4.1 trnsfrms.s 137
10.4.2 core_05.s 138
10.4.3 bss_05.s. 139
10.4.4 data_05.s 139

11. Flying Around The World 151
11.1 Introduction 151
11.2 Coordinate Transforms and Direction Cosines 153
11.3 Base Vectors and Direction Cosines 156
11.4 Rotating the Base Vectors: Rotation About an Arbitrary Axis 157
11.5 Accumulating Errors. 159
11.6 Clipping in 3D. 161
11.7 Velocity of the Observer 162
11.8 Example Programs 163
11.8.1 wrld_vw.s 163
11.8.2 core_06.s 163
11.8.3 bss_06.s. 164

12. A World Scene 176
12.1 A Database 176
12.1.1 A Map. 177
12.2 Sorting 178
12.2.1 A Bubble Sort. 179
12.3 The Viewing Transform 180
12.4 Running Times. 182
12.5 Example Program. 183
12.5.1 wrld_scn.s 183

12.5.2 data_06.s . . . 183
12.5.3 data_07.s . . . 184
12.5.4 data_08.s . . . 184
12.5.5 core_07.s . . . 184
12.5.6 bss_07.s . . . 185
12.5.7 systm_05.s . . . 185
12.6 Epilogue . . . 186

Appendix 1. 68000 Instruction Set . . . 209
A1.1 Registers . . . 210
A1.2 Addressing Modes . . . 210

Appendix 2. Devpac Assembler . . . 214
GENST . . . 214
The Editor . . . 215
Assembly . . . 216
Hunting for Bugs . . . 217

Appendix 3. Number Systems . . . 218
Binary . . . 218
Hexadecimal (hex for short) . . . 219
Negative Numbers . . . 219

Appendix 4: ST Operating System . . . 221
Calls to the Operating System . . . 221
BIOS calls (trap #13) . . . 222
XBIOS calls (trap #14) . . . 223
VT52 Terminal Escape Codes . . . 224

Appendix 5: Line A (A–Line) Routines . . . 226
Initialization $A000 . . . 227
Line A Variable Structure . . . 227
Line A Routines . . . 229

Appendix 6: Vectors and Matrices . . . 236
Vectors . . . 236
Matrices . . . 237
Products of Vectors . . . 239
The Scalar (Dot) Product . . . 239
The Vector (Cross) Product . . . 240
Surface Normal Vectors . . . 241
Base Vectors . . . 242
Matrices . . . 242
Homogeneous Coordinates . . . 242

Appendix 7: Geometric and Coordinate Transforms . . . 243
Coordinate Systems and Frames of Reference . . . 244

Appendix 8: Colour Palette and Key Scan Codes . . . 248
Standard Colour Palette . . . 248
GSX standard keyboard mapping . . . 249

1

An Overview

Computer graphics is not a minority interest of computer freaks. It is a multi-billion dollar industry. Even in 1982 when Hollywood spent 3 billion dollars on movie production, the world commercial computer graphics industry spent 2 billion dollars and was growing at the rate of 30% a year. In the same year in the U.S. 10 billion dollars were spent on video games. There has been no halt since that time. Computer graphics is very big business indeed.

The microcomputer owner meets some of the best graphics for his machine in games, many of which use advanced concepts straight out of the professional computer journals. For small machines there are always limitations on what can be achieved, determined by the speed of the processor and the size of RAM. But in recent years the popular microcomputer has been extremely good value for money, having considerable computational power at very low price and providing complex graphics at minimal cost. The Atari ST is just such a computer. So is the Commodore Amiga. This explosion in the power/price ratio of computer hardware has put immense computing capability in the hands of the popular micro owner and made advanced graphics techniques, which were the domain of the professional, available to anyone.

The aim of this book is to develop fast 3D solid graphics routines which run in real time and include features such as windowing (clipping), hidden surface removal, illumination from a light source, joystick control and full perspective and rotational transforms. The programs are written in 68000 machine code to run on an Atari ST but the algorithms are valid for any machine. In short, everything needed to get started on a flight simulator.

The programs are written in assembly language for maximum speed and have been tested and run using the Hisoft Devpac assembler. There are many excellent commercial assemblers available at modest expense, and even some in the public

domain. The Devpac assembler has been used here because it is excellent. There is nothing more irritating when looking for a persistent and obstinate bug in a program than an unfriendly assembler. The Devpac assembler has been a friendly and helpful companion through the many hours required to develop the programs in this book.

1.1 A New Medium

What is 'computer graphics'? It is certainly shrouded in mystique to some degree. Because it is still a relatively young subject its evolution is continuing apace, and is intimately linked to the power of current computers and the special graphics hardware incorporated in them. The solutions to many of the problems of yesterday, once based in software, are now provided at great speed in hardware. It is likely that much of the software of the kind developed in this book will be replaced in future machines by dedicated 'geometry engines'.

1.1.1 Is it Art, or What?

Humans are very good at generating and recognising complex visual patterns but not very good at doing arithmetic. By contrast, digital computers were designed to be perfect at binary arithmetic. What else they can do depends on how well complex mathematical functions can be constructed from basic binary arithmetic. There is a limitation here since numbers in a computer cannot be more accurate than the number of bits assigned to them but, apart from that, it is clear that complex mathematical calculations can be done quickly on even very modest microcomputers.

In computer graphics, the computer adds tremendous speed to any calculation associated with geometry, which is the mathematics of drawing. Because geometry is concerned with the exact mathematical relations between lines and surfaces, it is ideally matched to the way the computer works. This is the good and the bad news of drawing with computers: precise mathematical functions can be expressed graphically at lightning speed but making them look like natural objects requires considerably more work. In fact much of the effort in computer graphics is now concerned with 'messing up' the perfect but sterile images of geometry to make them fit for human consumption. Doing this has less to do with computers and more to do with the traditional skills of animation discovered many years ago by Walt Disney.

It is very easy to draw precise mathematical shapes with a computer because such shapes can be generated from a formula. A circle is an example of a simple mathematical function. For a circle centred at the the origin of an x-y coordinate system the formula is

$$x^2 + y^2 = r^2$$

Such a function is a good starting point for a billiard ball but a poor starting point for an apple, although superficially the difference is not all that great (both have an overall spherical shape with a shiny exterior). Let's consider how we might use a computer to draw an apple.

First of all there has to be a good starting point. There is no such thing as a mathematical formula for an apple. All apples are different. However, apples do have a typical shape and that is what the human artist knows from experience. But an artist would not draw all the apples in a still life with the same shape, it would be too boring. Programming a computer to avoid repetition and simplicity is difficult.

One way to draw apples would be to use equations of curves having the apple shape. By choosing functions with high powers of x, y and z, as much sharpness or flatness as desired can be included. This is the world of bicubic patches, Bezier functions and beta-splines. This would certainly allow variation, but with considerable computatational effort. One way to do this would be to hold different apple outlines as (x,y) coordinate pairs in a data base and then use curve and surface fitting techniques to connect then as in a "join the dots" picture. This is how the famous teapot of Martin Newell, which was a prototype in the early development of modelling solid surfaces, was constructed. In technical language it can be constructed from an outline consisting of three Bezier curves. Since the teapot is symmetrical, its surface (with the exception of the spout) is then generated by rotating the outline about the central vertical axis.

Another way is to avoid curves altogether, and instead subdivide the surface of the apple into many flat facets like a gemstone. By making the facets sufficiently small and numerous, an apple of any shape can be modelled. The little facets, being flat and many sided, are polygons and the surface of the apple is a polygon mesh. This approach is less time consuming than using curved patches but there remains the problem of disguising the sharp boundary edges between polygons.

This leads to the next level of refinement in producing a convincing image. A mathematical function on its own knows nothing of the laws of physics. These are so familiar to us that we take them for granted: glass is transparent but wood is opaque, metals look bright and shiny but human skin is dull and diffuse. Somehow these subtle but essential clues must be included. The most important first step is to make the rear surfaces of opaque objects invisible. This is called hidden surface removal which, despite the apparent simplicity of the task, turn out to be quite difficult. Much time has been spent investigating efficient and thorough ways of doing this. Next there must be visual clues to the surface structure. One obvious step is to illuminate it with a light source so that one side is brighter than the other.

At the next level of refinement the surface must be textured and patterned in a "natural" way to look real. In this the programmer is aided by the mathematics of fractals, developed and promoted by Benoit Mandelbrot. This is the geometry of

self-similar structures and quite different from the geometry of Euclid where structures are built from perfect lines and surfaces. Natural objects appear to have a lot in common with self similar structures and even if the similarity is not exact, they are convincingly modelled by them. A self-similar structure is one which has the same appearance at any level of magnification. Of course natural objects may only satisfy this definition over a limited range of dimensions but it often produces very convincing results. For example, the side branch of a fern when magnified looks like the main branch and small pebbles under magnification look like boulders. Nature is full of such structures. An additional bonus is that algorithms have been discovered which allow self-similar structures and landscapes to be generated from a relatively small amount of information. This relieves the programmer of carrying a colossal database from which to generate each separate detail of a complex scene.

All of these steps are essential to give a convincing image. The fact that so much visual richness is required to make an image look real testifies to the very advanced pattern recognition capability of human beings.

When all this is done, what have we got? Just a very roundabout way of painting an apple? The difference is that once created in software the graphic entity has an independent existence. The picture on the screen is just the final stage. Even if not being currently displayed, it can evolve according to rules included in the program. There is not even the constraint to create objects which are modelled on real life. It is possible to invent new "lifeforms" inside the computer. In Computer Aided Design (CAD) this is what happens all the time. Machines are designed, built and tested inside the computer long before they exist as material objects. In simulators and games this aspect is pushed as far as possible. Computer games specialise in generating artificial realities; the more exotic the better.

Future developments in input-output devices will undoubtedly have a major impact on what is currently called computer graphics. At the moment the emphasis is on generating realistic images. But images are only computer output designed for human input through the eyes. What will it be called when all of the senses are involved? Already, with the aid of spectacles which give separate input to each eye and tactile stimulators on the hands, it is possible to enter totally into the world inside the computer. What will it be like when computer couples directly into the human nervous system without the need for an intermediate interface?

Computer graphics is the thin end of a very long wedge which started when computers first produced a visual output in response to human input. Where it will end is unknown, but along the way it is sure to be lots of fun.

1.2 What Can You Do With A 16-bit Micro?

The answer to this question is best illustrated by looking at what is achievable on a powerful commercial system, of which a good example is the Reyes system developed at Lucasfilm Ltd and currently in use at Pixar. This has been used to make a number of well known short film sequences including "The Adventures Of Andre and Wally B", "Luxo Jr.", "Red's Dream" and the animated knight sequence from "Young Sherlock Holmes". The Reyes system was set up to compute a full length feature film in about a year, incorporating graphics as visually rich as real life. Assuming a movie film lasts about 2 hours and the film runs at 24 frames per second, this means each frame must be computed (rendered) in approximately three minutes.

The basic strategy in this system is to represent each object (geometric primitive) in a scene by a mesh of micropolygons which are subpixel-sized quadrilaterals with an area of $1/4$ of a pixel (the smallest visible unit on the screen). All the shading and visibility calculations are done on these micropolygons.The overall picture is constructed like a movie set with only the visible parts actually being drawn. Micropolygons are deemed to be invisible if they lie outside a certain viewing angle or are too close or too far away. The final system includes subtleties such as motion blurring, the effect whereby objects in motion appear to be blurred at their trailing edges. This is one of the devices used to enhance the impression of motion and is another lesson learned from traditional cartoonists.

A very complex picture in this system typically uses slightly less than 7 million micropolygons to render a scene of resolution 1024x612 pixels. With 4 light sources and 15 channels of texture a picture takes about 8 hours of CPU time to compute on a CCI 6/32 computer which is 4-6 times faster than a VAX11/780. Frames from "Young Sherlock Holmes" were the same resolution and took an hour per frame to compute, slightly more than the 3 minutes per frame aimed for. In the final movie all the stored frames are played back as in a conventional film. We can conclude that computational time is still a little too extended.

But it's not necessary to go as far as this to produce high quality pictures. There are now (1990) "personal" graphics stations available at prices almost within the reach of mortals. The Personal Iris machines manufactured by Silicon Graphics are good examples. They offer 256 colours (8 planes) from a palette of 4096 and, using a hardware "geometry engine", are able to perform transforms such as scaling, rotation, hidden-line removal and lighting, amongst others, to produce 3D motion in real-time. The CPU is a 20MHz R3000 RISC processor with a R3010 FPU (floating point unit). Here RISC technology has been used to maximise the speed, but it is interesting to note that before 1986 Silicon Graphics used the 68000 processor. It will not be long before machines such as these drop into the personal computer market.

What about a micro with 512 kbytes of RAM and a CPU working at 8 MHz? The potential for detailed graphics is somewhat less, especially if frames are to run in real time, sufficiently fast to avoid intolerable flicker. But it is surprising how much can be achieved. For speed, building up solid objects using polygon meshes is most attractive since it only requires that the vertices be stored, and a large object can be described by a very small amount of information. Moreover, since polygons are sets of vertices joined by straight lines, the most complex algebra involved will be that of simple geometry. This is the strategy we will use.

1.3 Assembled for Speed

There are many computer languages but assembly language gives the the best opportunity of getting as close to the hardware as possible and tailoring to the application in hand. All the programs in this book are written in 68000 assembly language and except for "housekeeping chores" and a few Line A examples in the first chapter, do not use any of the routines in the ST operating system. The programs could therefore easily be rewritten to run on a processor other than the 68000 since the most difficult thing is the overall program structure. Language details are secondary.

Assembly language is very exacting and unforgiving with a masochistic charm all of its own. It really has very little grammatical structure beyond the syntax of the instructions themselves, and the main criteria for efficient programming are speed, economic use of registers and memory, and efficient parameter passing. Sometimes there is conflict between these, especially where there is no shortage of memory. Where speed is all important, programs often sacrifice brevity in order to avoid time-consuming subroutine calls.

The programs in this book have been assembled and run using the very popular Devpac ST assembler from Hisoft. The Hisoft assembler has been used because it is powerful and friendly. It provides an excellent and relatively inexpensive workbench for program development. In particular the simple but powerful INCLUDE directive allows files to be pulled together at assembly time without the need to define global variables. The INCLUDE directive can be nested to any depth that memory will allow so that each chapter can INCLUDE the programs from earlier ones. In this way there is hardly any duplication, and a program file, once entered, can be used later. The overall program therefore grows steadily in size as the book progresses and practically no programming effort is wasted. The final program INCLUDE's all earlier parts. This is the only linking which needs to be done and it is painless.

Appendix 2 gives a brief description of assembler usage in general and the Devpac assembler in particular, including those commands which have been found to be most useful.

1.4 Writing for a 16 bit Micro

Writing programs in assembler for a 16 bit micro is quite different from writing for an 8 bit micro. Apart from the more powerful addressing modes available, there is a fundamental difference which centres on the ideas embodied in position dependent and independent code. The picture is somewhat confused by other similar sounding terms such as absolute and relocatable code. We shall discuss what these mean because they have a profound effect on how a program is written in assembler.

In an 8-bit micro usually only one program at a time is loaded in RAM and at a fixed location. Of course where an operating system oversees the running of programs, such as CP/M, things are more complicated. But in small micros with built in BASIC and very little else, the operating system reserves fixed space for its variables area and frees everything else for the current program. Knowing where the program resides in memory makes life simple for the programmer since fixed addresses can be assigned for variables and these will never change. A program which directly addresses fixed memory locations is said to be written in position dependent or absolute code.

Though such code can be written for computers with operating systems, there is another way of doing things which gives much greater flexibility, and allows several programs to reside in memory simultaneously. A consequence of this is that the actual position in memory of a particular program will not be known until run time. As a result, no actual actual numerical address can be referred to in the program since it is not fixed until the program is loaded and run.

There are several ways of overcoming this problem. One way is to use an addressing mode of the processor specifically designed to generate position independent code. This is called PC (program counter) relative addressing. What it does is locate an address not as an absolute value but relative to the value of the program counter where the reference is made. The assembled code will tell the processor to calculate the actual address by adding or subtracting a displacement to the current value of the program counter, which will always have a fixed value relative to the start of the program.

Another way is to calculate all addresses from a base address, or pointer, held in an address register. The program will then constantly refer to offsets from the address register but no actual value for the address need be specified when the program is being written. The register cannot, of course, be used for anything else while it is reserved in this way. The special register will have to be set up at the start of the program with the correct pointer. A good pointer is the address of the end of the program.

Another way is to allow the assembler take care of everything and generate relocatable code. This is code where no reference to specific addresses is made, but instead labels are used. The label name is chosen to be informative and of assistance to the programmer. For example, COLOUR might be the label for the long word address where the byte length value of the current colour of a polygon is held. The assembler will mark such a label as relocatable and its address will finally be fixed by the computer operating system when the program is loaded.

All of the programs in this book use relocatable code generated by the Devpac assembler. It is simple to write.

The instruction set of the 68000 is long and complex. To fully appreciate its power and elegance the reader should refer to the Motorola 16-Bit Microprocessor User's Manual. A brief listing is given in Appendix 1.

1.5 The Programs

The programs in this book have been written using the Devpac assembler and are ready to run. Once a program has been entered all that is necessary is to assemble it from within the editor and it will run as described. The program files all have the extension .s since they are source files. If a program is to run independently it can be assembled to disc with the file extension .prg.

The programs have all been run extensively to ensure they are as bug free as possible, and the listings have been obtained from within the assembler Editor using the PRINT BLOCK facility to ensure that there are no further stages of transcription during which errors might creep in. However as with all human endeavours, there can be no guarantee that the programs are completely bug free.

The programs are undoubtedly neither the fastest nor most elegant examples of their kind in existence but, in a tutorial of this kind where the emphasis is on teaching, the main point is to understand how things are done. The astute reader will quickly discover clever ways of improving them. In any case the best commercial programs are proprietary and kept secret from us.

1.6 The ST Operating System

The ST operating system is large and complex and operates at many levels. There are often many ways of doing the same thing depending on the level of entry.

At the top are the device-independent parts: VDI (virtual device interface) and AES (applications environment services). In the middle are the device dependent parts: BIOS (basic input-output system), XBIOS (bios extension), BDOS (basic disc operating system). Collectively, these middle level calls are called GEMDOS. At the bottom are the A-Line or Line A routines which provide a fast interface to

the low-level graphics primitives as provided in all ST computers with TOS in ROM. TOS is the name given to the overall Operating System.

Using the device independent routines ensures that programs are portable, i.e. they are shielded from hardware details and in principle work on any machine with the same operating system. The penalty is one of speed. Generally the closer you get to the hardware, the faster things run. Using the Line A routines for speed means that the programs are not portable to other machines. This is not important if the programs are being written exclusively for the ST.

In this book we will occasionally use BIOS, XBIOS and Line A routines. In particular they are used to make "legal" calls to the operating system, particularly where system variables addresses are required in order to make the programs "future proof" against low level modifications at some later time.

Apart from this all the programs are "original" (if there is such a thing in programming) and tailored closely to the graphics applications.

2

Drawing on the Screen

In this chapter we look at how the ST screen is addressed. This is detail which is highly specific to the ST but of great importance for fast graphics, since our intention is to bypass the routines of the operating system and draw 3D solid objects in real time. A very important aspect of this will be filling in polygonal shapes quickly.

No matter how complex graphics programs are, ultimately their output must appear on the screen. Actually, to be precise, on the physical screen. There is a distinction between the physical screen, which is that part of RAM which holds the picture frame currently being displayed on the monitor, and the logical screen, which is where the output of the program is currently directed. These are just two 32 kbyte blocks of RAM and the distinction between them is that the hardware thinks it should display the one called the physical screen. To produce flicker-free graphics, the usual scheme is to alternate the names of these two screens so that one of them is being displayed whilst the other is being drawn. This is often called 'screen buffering'. The switch from one to the other is naturally synchronised to the program; the program doesn't ask for the switch until the new frame is complete and the hardware doesn't change the display until the raster on the screen has reached the bottom right-hand corner and is ready to fly back to the top. The short time for this to occur - called the vertical blank - is more than sufficient for the hardware to switch the screens. There is even a routine in the operating system which will do all this in one go. To start with, for simplicity, we will use only one screen. The switch to two screens is not difficult.

2.1 The Screen

What then is a screen? Well this depends on the resolution. All resolutions have one thing in common though. They all consist of a single block of RAM 32 kbytes

in length. How this is used is quite different in the different resolutions. Of the three resolutions, high, medium and low, it is really only the low resolution which offers the extensive use of colour. High resolution is very poor indeed for colour - there isn't any, but it does give very clear pictures in monochrome. Medium resolution is somewhere in between but is not widely used.

To understand the problem, think of the differences between the actual monitor screen and the block of RAM holding the image. The actual screen is a rectangular end of a cathode ray tube on which an electron beam writes. To make this look like a picture the beam moves very quickly from left to right and top to bottom in a series of 'raster' scans; the picture is made up of closely spaced horizontal lines. There isn't really a solid picture at all, it just looks that way from a distance. Memory, on the other hand, is laid out as a contiguous line of bytes, which are the smallest elements the microprocessor can directly address. Of these, the smallest resolvable unit is the bit (8 bits = 1 byte). Somehow each bit in memory must directly relate to the smallest 'spot' or pixel on the screen. Let's look at how this is done in high resolution which is the simplest case.

2.1.1 High Resolution

In high resolution there is a very simple relation between each memory bit and each screen pixel. Figure 2.1 shows the 'mapping' from one to the other. Very simply, if a bit in memory is set then a screen pixel is on. There are 640 pixels horizontally and 400 vertically. Multiplying these two numbers and dividing by eight gives the total number of bytes in the screen RAM, 32000. The 68000 does

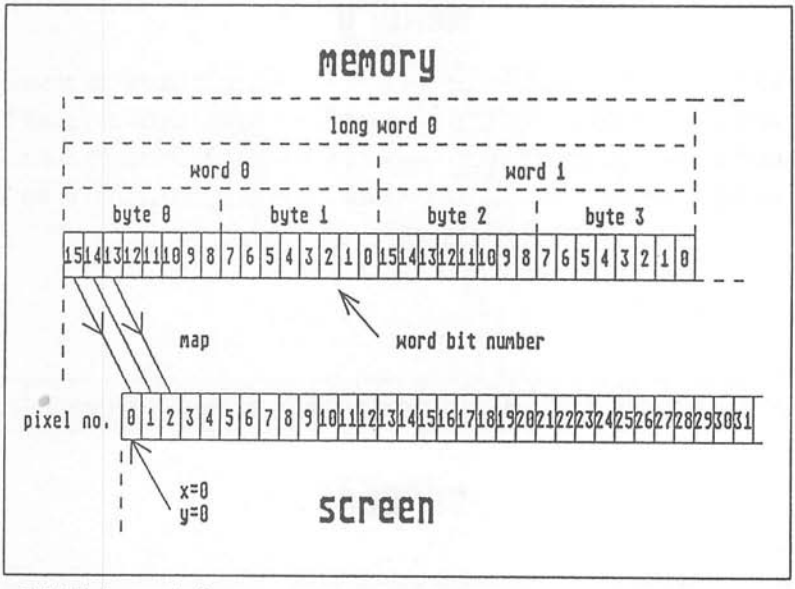

Figure 2.1 High resolution screen memory map

not directly communicate with bits however. It can directly address bytes and groups of them: words (1 word = 2 bytes) and long words (1 long word = 2 words = 4 bytes). Note also that the origin of the screen coordinates is at the top left-hand corner and how the bit number in each word in memory increases to the left whereas the pixel position increases towards the right. Although the smallest addressable unit of memory is the byte the designers of the ST decided to use the word as the basic building block in screen RAM. In high resolution this is hardly important but in low resolution, the word rules, O.K.

2.1.2 Low Resolution

In low resolution, things get complicated because of colour. The problem is that each pixel can have one of the 16 different colours currently set in the colour palette. This does not mean that there are only 16 specific colours available. It means that at any given instant only one of the palette colours can be selected. While no change is made to the colour palette the colours are fixed, but the palette can be altered, at a convenient moment, to contain 16 new colours out of a maximum of 512. A good time to do this is during the fly-back of the electron beam from the bottom right-hand corner of the screen to the top left-hand corner (the vertical blank) and it is even possible to change it during the short time it takes for the fly back from the end of one line to the beginning of the next (the horizontal blank) so that up to 512 colours can be displayed simultaneously.

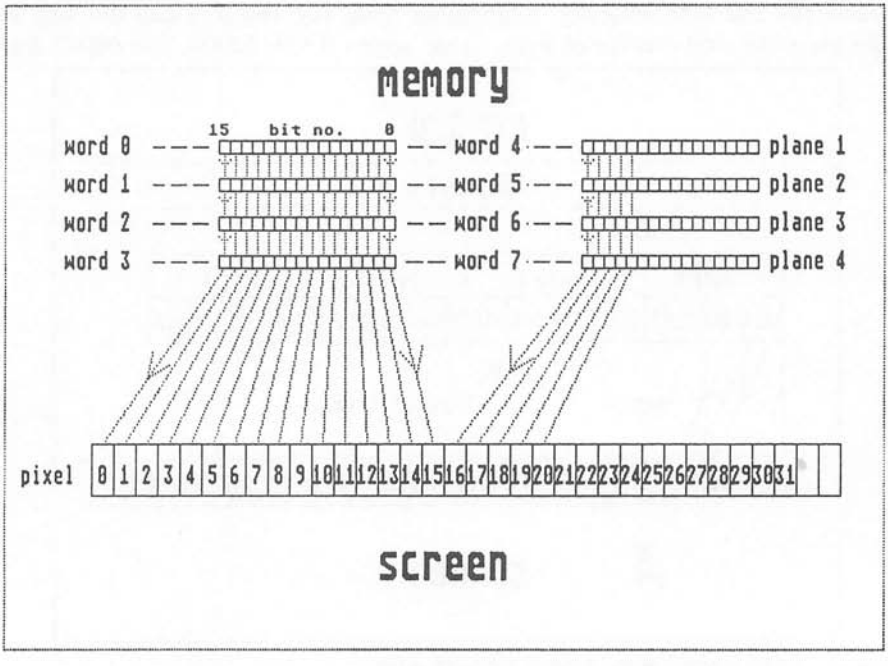

Figure 2.2 Low resolution screen memory map

Somehow then the pixel colour, as a number between 0 and 15 must be specified. This has been done in a way which keeps the total screen RAM fixed at 32000 bytes. Obviously there has been a trade-off of colour against resolution. A number between 0 and 15 can be written in a nibble (4 bits) and this is the key to how the screen RAM is laid out in low resolution. Each bit in this nibble is referred to as a 'colour plane'. So the first word of memory is the first colour plane, the second word is the second colour plane, the third word is the third plane and the fourth word is the fourth plane of the first 16 pixels. Then the pattern is repeated for the second 16 pixels and so on. In this way each pixel has a nibble all to itself to specify its colour. This arrangement is shown in figure 2.2. The need to reserve a nibble for the colour means that the overall number of pixels that can be written is reduced to one quarter of that in high resolution. Hence the halving of both the x- and y-resolutions to 320 by 200 respectively. For the moment we will use the ST's standard palette. Later on we will change it to simulate different levels of illumination. A list of the colours of the standard palette is given in Appendix 8.

2.1.3 Medium Resolution

Medium resolution is not of much interest to us but it uses memory in a way somewhat intermediate between that of high and low resolution. There are two colour planes which means that 4 colours can be specified (numbers 0 to 3). The penalty is that the number of pixels is halved to give screen resolutions of 640 and 200 for x and y respectively. The screen looks rather squashed in the x-direction.

2.2 The Line A routines

The ST comes armed with an arsenal of built-in graphics routines. Some of them are very useful. Others look useful but are not very fast. The example programs included in this chapter illustrate the use of two of them. They are called Line A routines because the instruction word which triggers them has the value *$a00n* in hexadecimal, where n is the number of the routine. The routine which draws horizontal lines, for example, has the code *$a004*. It is the fourth routine in the list. The way these work exploits a design feature of the 68000 processor in which unusual (or non-legal) and illegal instruction codes, which on other processors might lead to a system crash, are brought within normal operation by calling them 'exceptions'. The 68000 recognises exceptions for what they are and has a special way of dealing with them. The Line A instructions cause an exception because of the hex $a which starts the code. When this is spotted by the processor it immediately jumps into supervisor mode and uses the "n" at the end of the word as an index to find which routine to jump to. When the routine has been executed control returns to the user. Such routines are like subroutine calls but much safer for the system since they are processed in supervisor mode and can't easily be meddled with.

The advantage of using them is that they come 'ready made' and take care of all the messy details like converting screen coordinates to bits to set in screen RAM appropriate to the current resolution. In a sense they make the graphics more 'idiot proof' since programs can't crash on basic technical details. The disadvantage is that because they must be very flexible they "worry" about too much and are consequently slightly slower than routines dedicated to a particular resolution. If only one screen resolution is being used, they can be tailored to this and speeded up a bit. The program examples at the end of the chapter illustrate their use in plotting pixels and drawing lines. The line A routines offer a wide range of options and although they are not used in this book, their potential is such, particularly in sprite graphics, that they are discussed extensively in Appendix 5.

2.3 Example Programs

The example programs included in this chapter explore simple drawing operations direct to the physical screen: plotting a point and filling a polygon. These are done in two different ways: either using Line A routines or using customised software. It is not claimed here that these programs are the fastest possible. Other versions may be more elegant and faster. But these programs are fast and do the job adequately. Besides, they do have an educational value, illustrating various aspects of assembly code programming. When you have studied how they work you are encouraged to make your own improvements.

Here is a brief outline of each self contained program, with a discussion of its salient features. The programs themselves are listed on the succeeding pages. They are ready for assembly by DevpacST. If another assembler is used, modifications to the syntax may be necessary as specified in its manual.

2.3.1 *put_pixl.s*

This program illustrates the use of the *$a001* routine to plot a point. It is "quick and dirty" in that no attempt is made to clear the screen and there is no orderly exit from the program. It just keeps plotting the same point over and over and to stop it you will have to switch off the computer. It is set up this way to show how little preparation is needed to produce an output on the screen. If your monitor isn't too good you may have to peer rather hard to see the dot.

The routine itself establishes what resolution is currently being used. As with all Line A routines, it must be preceded by the system initialisation call to *$a000*. This returns the addresses of the table of routines and the location of the variables table to which parameters must be passed. This table also itself contains addresses of further tables, the *PTSIN* and *INTIN* arrays. A further discussion of these arrays is give in appendix 5.

2.3.2 *set_pixl.s*

This is the equivalent of *put_pixl.s* but with the routine *$a001* replaced by a low resolution screen driver routine. The convenience of using *$a001* is clear. This program spends much of its time converting the x and y coordinates of the pixel to a bit location in memory and is only valid in low resolution. In addition it must first find where the physical screen starts in memory which it does with a call to that part of the ST's operating system called the XBIOS. The various parts of the operating system are discussed in greater detail in Appendix 4.

All 'housekeeping' subroutines, including calls to the operating system are contained within a separate file called *systm_oo.s*. This must be present when the program is assembled so that it can be *INCLUDE*d at that time. This is accomplished by the assembler directive *INCLUDE* which appears at the top of the listing. A fuller explanation of the Devpac assembler is given in Appendix 2. Another housekeeping subroutine included in *systm_oo..s* is *wrt_phys_tbl*. This is used to avoid a multiplication in finding the address in RAM of the start of the row corresponding to the current y-coordinate. Since this is a calculation that must be done each time, it makes sense to do the work beforehand and record the results in a 'look-up' table which can easily be accessed using the value of y as an index. Look-up tables are frequently used to avoid multiplications and divisions during the program. These are among the most time-consuming instructions in the set.

2.3.3 *systm_00.s*

This is the general housekeeping file referred to in the previous section. It will not assemble and run on its own but is meant to be included in other programs at assembly time. The directive *INCLUDE* takes care of this. The file contains frequently used subroutines of a utility nature; at present it contains three routines:

find_phys - find the address of the start of the physical screen,

wrt_phys_tbl - write a look-up table of the screen row addresses

hline_lu - write a look-up table of masks for horizontal line drawing.

The first of these uses an Operating System call to an XBIOS (extended BIOS) routine to find the starting (base) address of the 32kbyte section of RAM, the physical screen, the contents of which are being displayed on the screen. The call uses one of the exception modes triggered by the *TRAP* instruction.

The second lays out a table of long word start addresses of the 200 raster scan screen rows, one for each value of the y-coordinate in low resolution. This helps to speed up drawing.

The last routine is also discussed later in the description of *polyfill.s*. It also sets up a table which is used to speed up drawing but in this case the table contains a

set of bit patterns or "masks". These masks are a block of 1's to set within a word when a line is to be drawn between two pixels.

At present there are only three housekeeping routines, but more will be added later.

2.3.4 *data_00.s*

This also is not meant to be assembled on its own, but is another *INCLUDE* file to be added in at assembly. It contains lists of permanent data, mostly the assembler directive *EQU* which allows us to replace Line A variable offsets by informative labels in the code and thus make it more readable.

2.3.5 *polyfil0.s*

This program illustrates the repetitive use of the *$a004* horizontal line drawing routine to fill a polygon, although the polygon is simply a rectangle here. This is obviously important since in solid 3D graphics a lot of time is spent shading in the polygonal surfaces of objects. Also it introduces the method we will use to feed data to the horizontal routine. There are many ways of doing this but we will calculate ahead of time the start and end x-coordinates of each scan line to fill in the polygon and store the data in a buffer which starts at the address labelled *xbuf*. How this is done for a polygon which is not a rectangle is another story to be explained later, but for the moment we can use it to see how fast a polygon can be filled. As before, the Line A routine works out all the details concerning the screen, and life is quite simple.

2.3.6 *polyfil1.s*

In this a polygon is filled as in the previous program, but the Line A routine is not involved. Instead the screen is directly addressed, so much of the program is concerned with taking care of screen details. Once again a look-up table is used to avoid a repetitive determination of a mask which is used in the horizontal line drawing subroutine. It isn't really necessary to use such a table since the result can be arrived at speedily using shift instructions, but it falls in with the philosophy of relegating awkward calculations to look-up tables. The task which it performs is to produce the pattern of bits which must be set to draw sections of a horizontal line within words and is used in the following way.

All horizontal lines are split into three sections: the start of the line to the end of the first word in screen RAM, a run of completely filled words and from the end of the last filled word to the end of the line. Hence there may be incompletely filled words at the start and end of the line; this is where the mask look up table is used. It quickly finds the pattern of bits to put in these words. The run of filled words is easy to handle (of course we mean filled by the colour; some colour planes may have all zeros). The line routine chops lines up into these categories and deals with each one separately.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                        put_pix1.s                                     *
*                                                                       *
* A program to plot a pixel using the Line A routine number $a001      *
* with the minimum of fuss. Pull the plug to stop it.                   *
* The system takes care of the screen resolution.                       *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

        SECTION TEXT

put_pix:
* Initialization: get the pointer to the Line A Variable Structure.
        dc.w    $a000           set up the Line A table pointer
* The address of the table is in a0. Get the pointers to the arrays.
        move.l  8(a0),a4        pointer to intin array
        move.l  12(a0),a5       pointer to ptsin array
* Enter the x and y coordinates of the pixel in screen coordinates.
        move.w  #200,(a5)       x = 200, first ptsin word
        move.w  #100,2(a5)      y = 100, second ptsin word
* Set the colour number - only the lowest bit works in high resolution.
* In low resolution this is the standard palette no. for red.
        move.w  #1,(a4)         colour red, first intin word
* Set the pixel
        dc.w    $a001           Line A Put Pixel
        bra     put_pix         repeat

        END
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                          polyfil0.s                                     *
* A program to fill a polygon using the Line A routine $a004 .           *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

        SECTION TEXT

main    bsr     poly            fill the buffer with x coordinates
        bsr     a004            use the x coords to draw the lines
        bra     main            keep going to cover the mouse

* A subroutine to fill the buffer with x values 16 and 256 to scan fill
* a rectangle between the limits x=16 to x=256 and y=50 to y=150.
poly    lea     xbuf,a0         point to start of buffer
        move    #50,d0          initial y=50
        lsl     #2,d0           *4 for the offset into the table
        adda.w  d0,a0           pointer to the initial long word
        move.l  #$00100100,d0   high word=16,low word=256
        move    #100-1,d7       fill 100 lines (up to y=150)
poly1   move.l  d0,(a0)+        fill the next long word
        dbra    d7,poly1        for all the y values
        rts
* Initialise the  a-line parameter block. Find its address.
a004    dc.w    init            returns the address in a0
* Set the constants for a horizontal line
        move.w  #1,fg_bp1(a0)   Set
        clr.w   fg_bp2(a0)      for
        clr.w   fg_bp3(a0)      mono-
        clr.w   fg_bp4(a0)      chrome
        clr.w   wrt_mod(a0)     overwrite.
        lea     fill,a2         Here is
        move.l  a2,patptr(a0)   the fill pattern
        move.w  #4-1,patmsk(a0) consisting of 4 lines.
        clr.w   multifil(a0)    The pattern is for one plane.
* Fill a line at a time using the $a004 routine.
        lea     xbuf,a1         pointer to base address
        move    #50,d1          start at y=50
        move    d1,d2           save it
        lsl.w   #2,d1           y*4 is the offset into the table
        adda.w  d1,a1           here is the first line
        move    #100-1,d7       draw 100 lines (counter is 1 less)
        subq    #1,d2           reduce initial y
        move.w  d2,y1(a0)       for the loop
* Here is the loop which fills each scan line in succession
poly2   addq.w  #1,y1(a0)       next y in the loop
        move.w  (a1)+,x1(a0)    next x1
        move.w  (a1)+,x2(a0)    next x2
        movem.l d7/a0-a1,-(sp)  save the registers
        dc.w    hline           draw the line
        movem.l (sp)+,d7/a0-a1  restore the registers
        dbra    d7,poly2        repeat for all y values
        rts

        SECTION DATA
        include data_00.s       add in the data file

        SECTION BSS
xbuf    ds.l    100             all the scan line x-coords.

        END
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                          polyfill.s                                   *
* A program to fast fill a polygon. The start and end x coordinates    *
* of each horizontal line are the high and low words stored at xbuf.*
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

        SECTION TEXT
        opt     d+              put in labels for debugging
        bra main                don't try to execute the include
        include systm_00.s      the housekeeping file

main    bsr     find_phys       locate the physical screen
        bsr     wrt_phys_tbl    where the rows start
        bsr     hline_lu        the masks for filling words
        bsr     poly            set up the buffer and fill it
        bra     main            keep going to cover the mouse

* Fill the buffer from y=50 to y=150 with the values 16 and 256 to
* fill à rectangle between the limits x=16 to x=256 and y=50 to y=150.
poly    lea     xbuf,a0         point to start of buffer
        move.w  #50,d0          initial y=50
        lsl.w   #2,d0           *4 for the offset into the table
        adda.w  d0,a0           pointer to the initial long word
        move.l  #$00100100,d0   high word=16,low word=256
        move.w  #100-1,d7       fill 100 lines (up to y=150)
poly1   move.l  d0,(a0)+        fill the next long word
        dbra    d7,poly1        for all the y values
        lea     xbuf,a1         pointer to base address
        move.w  #50,d1          start at y=50
        move    d1,d3           save it
        lsl.w   #2,d1           y*4 is the offset into the table
        adda.w  d1,a1           here is the first line
        move.w  #100-1,d7       draw 100 lines (counter is 1 less)
        subq    #1,d3           reduce initial y
poly2   addq    #1,d3           next y
        move.w  (a1)+,d2        next x1
        move.w  (a1)+,d1        next x2
        sub     d2,d1           x2-x1
        addq    #1,d1           N= no to do
        moveq   #1,d4           system colour #1 - red
        lea     phys_tbl_y,a4   where the screen starts
        movem.l d0-d7/a0-a6,-(sp)       save all registers (why not!)
        bsr     holine                  draw the line
        movem.l (sp)+,d0-d7/a0-a6       restore the registers
        dbra    d7,poly2        repeat for all y values
        rts

*HOLINE. A horizontal line is drawn from left to right.
* passes: x1=d2.w, y1=d3.w, N=d1.w, colour=d4.w, phys-screen:a4.l
* First find the address of the word at which the line starts.
holine  lea     hln_tbl,a3      pointer to mask table
hline0  lsl.w   #2,d3           there are y long words before the
        movea.l 0(a4,d3.w),a4   current row address in the table
        move    d2,d5           save x1
        andi    #$fff0,d5       go in steps of 8 bytes
        lsr.w   #1,d5           to point to plane #1 word
        adda.w  d5,a4           at this address
        andi    #$000f,d2       which pixel from the left?
        move    d2,d0           save it
```

```
* does the entire line lie within one word?
        subi    #16,d0
        neg     d0              are there more pixels to the word end
        cmp     d1,d0           than we have to draw?
        bmi     long_line       no, so it's a long line
* The line is entirely within one word. Get the mask and draw it.
        move    d1,d0
        bsr     draw_it
        rts                     and that's all.
* complete 1st word in a long line
long_line:
        sub     d0,d1           number left
        bsr     draw_it
* Now fill all the solid words.
hline6  clr     d0
        not     d0
        move    d1,d2           save number of pixels left to do
        lsr     #4,d2           how many are whole words?
        beq     last_word       none are
* a long stretch of filled words - no need to read the table
        subq    #1,d2           this many full words but one
        move    d0,d3           which are all 1's
        not     d3              or all 0's, depending on the colour
        moveq   #4-1,d5         4 colour planes
        move    d4,d6
        subq    #2,a4
inc_plane:
        addq    #2,a4           offset for next plane
        movea.l a4,a5           save the address
        move    d2,d7           initialise the word count
        lsr.w   #1,d6           next colour bit
        bcc     clr_word
set_word:
        or.w    d0,(a5)
        adda    #8,a5           next word in this plane
        dbra    d7,set_word
        bra     new_plane
clr_word:
        and     d3,(a5)
        adda    #8,a5           next word in this plane
        dbra    d7,clr_word
new_plane:
        dbra    d5,inc_plane    for all the colour planes
        subq    #6,a5           pointer to next plane 1
        movea.l a5,a4           update pointer
* it only remains to do the last word. It will start at pixel 0
last_word:
        andi    #$f,d1          low nibble
        cmpi.w  #0,d1           any to do ?
        beq     holine_end      no - finished.
* In finding the mask,the row offset is zero this time.
        clr     d2              1st pixel at extreme left
        move    d1,d0
        bsr     draw_it
holine_end:
        rts                     completely finished
```

```
* Draw in a section of a word which starts at pixel a and ends at pixel b
draw_it
        lsl     #5,d2           the mask row offset=a*32
        move    d0,d5           plus
        subq    #1,d5           column
        lsl     #1,d5           offset of (15-b)*2 gives
        add     d5,d2           the total offset
        move.w  0(a3,d2.w),d0   to fetch the mask
        move    d0,d3           and
        not     d3              its 1's compliment
        moveq   #3,d5           (4-1) colour planes
        move    d4,d6           save colour
next_plane:
        lsr     #1,d6           is this colour bit set?
        bcc     not_set         no
        or.w    d0,(a4)+        yes, also set the bits
        dbf     d5,next_plane
        rts
not_set and.w   d3,(a4)+        clear the bits
        dbf     d5,next_plane
        rts

        SECTION BSS
xbuf    ds.l    400             the buffer of x word pairs
phys_screen     ds.l    1       the address of the physical screen
phys_tbl_y      ds.l    200     pointers to the row y's
hln_tbl         ds.w    256     the masks for filling words
        END
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                        data_00.s                                    *
*                                                                     *
*                      The data file                                  *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* A list of the Line A variables offsets.
* Their meanings are given in Appendix 5.
fg_bp1  equ     24
fg_bp2  equ     26
fg_bp3  equ     28
fg_bp4  equ     30
wrt_mod equ     36
x1      equ     38
y1      equ     40
x2      equ     42
patptr  equ     46
patmsk  equ     50
multifil        equ     52

* and the routine names
init    equ     $a000
hline   equ     $a004
* The fill pattern for the $a004 routine
fill:
        dc.w    %1111111111111111
        dc.w    %1010101010101010
        dc.w    %1001100110011001
        dc.w    %1111111111111111
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                       *
*                        systm_00.s                                     *
* Calls to the operating system and frequently used subroutines.        *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
find_phys:
* A call to the operating System to find the physical screen address
        move.w  #2,-(sp)        xbios _physbase
        trap    #14             xbios call
        addq    #2,sp           tidy stack
* the base address is returned in d0 and saved
        move.l  d0,phys_screen
        rts
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
wrt_phys_tbl:
* Write a look-up table of the addresses of the start of physical each
* screen row in low resolution. The product 4*y is an offset to row y.
        move.l  phys_screen,d0  where screen location is kept
        move    #200-1,d1       200 rows
        lea     phys_tbl_y,a0   where the table is
luloop  move.l  d0,(a0)+        the next row in the table
        add     #160,d0         there are 160 bytes/row
        dbra    d1,luloop       for all rows
        rts
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
hline_lu:
* Set up a look-up table for low resolution horizontal line drawing.
* Each mask in the table is the word to set between pixel a and b.
        lea     hln_tbl,a0      pointer to the table base
        move.w  #16-1,d1        16 rows, d1 is the counter
hloop2  clr.w   d0              new row
        move.w  #16-1,d2        16 columns, d2 is the counter
        bset    d1,d0           set the 1st column bit
hloop3  move.w  d0,(a0)+        next column
        move.w  d0,d3
        lsr.w   #1,d3   shift
        or.w    d3,d0   add back
        dbra    d2,hloop3       complete this row
        dbra    d1,hloop2       for all rows
        rts
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                        set_pix1.s                                     *
*                                                                       *
* A program to set a pixel in low resolution using a low                *
* resolution screen driver. The Operating System is used to find        *
* the address of the physical screen. Also a look-up table is           *
* constructed to quickly find screen row adreses from y coords.         *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        opt     d+                  write in labels for debugging

        SECTION TEXT
        bra     main                don't try to execute the include
        include systm_00.s          include the file of subroutines

* Here is the main program
main:
        bsr     find_phys           locate the physical screen address
        bsr     wrt_phys_tbl        construct the row address look-up
        move    #160,d0             plot the pixel at x=160
        move    #100,d1             and y=100
        moveq   #1,d2               with colour red
        bsr     set_pix             plot a point
        bra     main                avoid being covered by the mouse

set_pix:
* The subroutine to plot a pixel at x,y in low resolution
* Entry: x=d0.w,y=d1.w,colour=d2.w. Corrupted: d0,d1,d2,d3,d4,d7,a0.
        lea     phys_tbl_y,a0       the screen base address is here
        lsl     #2,d1               4*y is the row offset
        movea.l 0(a0,d1.w),a0       to this row address
        move    d0,d5               save x
        andi    #$fff0,d5           go in steps of 4 words
        lsr.w   #1,d5               to the first word in the group
        adda.w  d5,a0               at this address
        and     #$000f,d0           this is the pixel number in the word
        subi    #15,d0
        neg     d0                  this is the bit
        clr     d1
        bset    d0,d1               this is the mask
        move    d1,d3
        not     d3                  and its complement
        move    #4-1,d7             the counter for the 4 colour planes
* Use the colour nibble to set the four colour planes
next_plane:
        lsr     #1,d2               is this bit set?
        bcc     clear_bit           no
        or.w    d1,(a0)+            it is, so set the plane
        dbra    d7,next_plane       for all planes
        rts
clear_bit:
        and.w   d3,(a0)+            the bit is zero so clear the plane
        dbra    d7,next_plane       for all planes
        rts

        SECTION BSS
* Where uninitialised data (that calculated by the program) is stored
phys_screen     ds.l    1           the address of the physical screen
phys_tbl_y      ds.l    200         pointers to the rows in low resolution
hln_tbl         ds.w    256         (not used just yet)
        END
```

3

Modelling a 3D World

One of the most fascinating things aspects of computers is the way they can be used to build life-like models. The great attraction of realistic computer games and, at the more serious end, simulators stems from the way the computer screen can be made to look like a window onto an invented universe. Some famous scientists, impressed with the similarity to the process of creation, have even gone so far as to consider theories of reality based on a real Universe built up from 'bits' of information. Whatever the fundamental significance of it all, the fact remains that computers offer a new dimension for human expression and experience. Simply put, they provide the possibility to create alternative realities where the laws of Nature may or not apply. All sorts of strange and exotic situations can be invented and investigated. For human beings, who relate most easily to objects and situations met in everyday life (and dreams), what appears on the computer screen should look familiar. Great effort has gone into constructing models of this kind. In a simulator which is supposed to accurately depict reality, the emphasis is on models which obey the laws of Nature precisely.

In this chapter we will look at a way of modelling which provides a very fast and reasonably accurate picture of real objects. For the most part, but not completely, this involves polyhedral structures with polygonal faces as the building blocks, the so-called 'vector' graphics. Spheres and other objects with a high degree of symmetry can also be drawn quickly. Actually, to set the record straight, vector graphics originally meant something else. It was a name given to a mode of display where points on the monitor were joined directly by an electron beam that could be switch quickly from one part of the screen to another. This did not require much memory devoted to the screen and gave very fast 'wire-frame' pictures. The displays on monitors today do not use this technique. Instead, the image is built up from horizontal raster scans from one side of the image to the other. It is called raster scan (or scan conversion) graphics and we have already

used it to fill polygons. The speed with which an outline can be filled by raster scans makes it a very useful technique. However the name vector graphics has become commonly used to describe the graphics modelling technique itself, not the display technology. The adjective "vector" here really refers to the extensive use made of vector geometry in the programs.

One other important technique is the BITBLT (Bit Block Transfer) type of graphics, in which SPRITES play an important role. The Atari STE has a piece of hardware on board, the BLITTER, which handles such operations very quickly, whilst on the 520ST it is done entirely by software. In BITBLT graphics, blocks of memory are manipulated as a whole, which is very useful since, once laid out in RAM, scan conversion need not be done a second time. The block of bytes is simply moved to the screen area. Some very clever and fast things can be done this way, particularly with sprites, but the relationships between the parts of the image are essentially determined by how the block is initially laid out in RAM. Sprite graphics is not discussed any further in the main text of this book, but Appendix 5, which lists all the Line A routines, contains an extensive coverage of the powerful sprite routines contained within the ST operating system.

Having said that, it is likely that the next generation of popular computers will have hardware implementation of all the common graphics functions including the 'vector' graphics we are about to discuss. It is very probable that soon all graphics functions will be done by very fast hardware 'geometry engines'.

3.1 3-D Modelling

"Real-time" 3D modelling has to be very fast. This is because humans can spot the flicker of the picture if it changes more slowly than about once every 50 milliseconds. In order to work in real time, the viewer has to be able to enter new data through the keyboard, joystick or mouse and see its effects immediately. The solid 3D structures which can be transformed and drawn on this time scale most easily are polyhedra.

Polyhedra are very good graphics building blocks or 'primitives' for several very good reasons:

- they are completely defined by their vertices,
- the faces are polygons with straight edges,
- in any transformation only the vertices need to be recalculated,
- a transformed polygon is also a polygon
- polygons can quickly be filled in to look 'solid' using raster scans.

What all this means really is that it's very hard to draw and shade in curved surfaces which don't have high symmetry (like circles) and the only 3D objects without curved surfaces are polyhedra.

In fact computer graphics does not have a monopoly on the use of polyhedra as basic building blocks. The real world uses them extensively; all houses are made from bricks, which are six-sided polyhedra.

3.2 Transformations and Frames of Reference

All of the above statements concerning polyhedra can be translated into a definite mathematical framework called vector algebra, which is a very elegant and precise formulation of the mathematics of lines and planes. It becomes even move useful when presented in matrix form and it is this approach which usually appears in text books on computer graphics. For someone with little knowledge of advanced mathematics this looks very intimidating. Actually it's not. Many secondary school syllabuses handle simple rotations using 2x2 matrices, and it really isn't much more complicated than that. For those of us who do not wish to blaze new trails in the world of mathematics it is simply a case of understanding the general method and taking the results on trust. After all, once you've seen the transforms working you can use them in your programs and forget about them. There's no need to re-invent the wheel.

For the moment though, in order to see the problem laid out in its entirety, let's consider all the various stages of transforms, as shown in figure 3.1. The distinction between the view frame and the world frame, and transformations between them, is discussed in further detail in Appendix 7.

3.2.1 The Object Frame

An object which exists inside the computer has quite a complicated life before it is seen on the screen. Most of this complication arises from the various transforms required to make it 'lifelike'. But whatever they are (rotations, translation or even something more exotic), the object must preserve its original identity, i.e. its relative dimensions. What this means is that no calculation can be absolutely precise and, with the picture being recalculated faster than 20 times each second, if the original definition were not continually referenced, it would not be long before accumulative errors would make it unrecognisable (this problem crops up in all our calculations which, for speed, are done in only limited accuracy). Therefore it is necessary to constantly refer back to the original data which define the object. We call this place, in which the object is defined, the object frame (there is nothing sacrosanct about this name, other people have invented other names). Of course it doesn't 'exist' in any real sense, it's just that the numbers which fix the positions of the vertices are coordinates measured from some origin. This origin is where the

object frame is said to be located. The object frame can be positioned so as to reflect the symmetry of the object. For example, the natural object frame of a cube could be a cartesian (x,y,z) coordinate system centred at the centre of symmetry (centre of gravity) of the cube, with the sides of the cube parallel to the x, y and z axes of the coordinate system as shown in Figure 3.1.

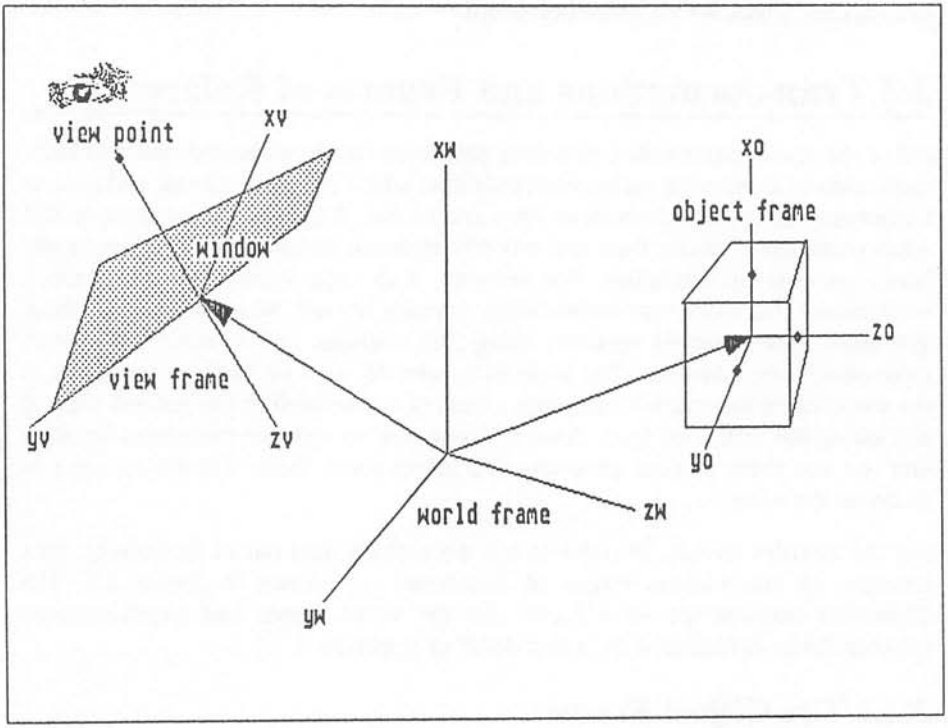

Figure 3.1 Frames of reference

There may be several object frames combined together, particularly when a complex object is made up of several simpler objects. The process of sticking together simple objects (primitives) to make a complex one involves just the kind of transforms we have been talking about. These transforms are sometimes referred to as instance transforms.

3.2.2 The World frame

Having constructed a complex object - which can be thought of as an 'actor' in the scenario we are about to create - it is necessary to place it in the arena with all other 'actors'. This common space, inhabited by all objects is called the world frame. It is the place where the Laws of Nature play a role. For example, objects which are not subject to any force either remain at rest or move at constant

velocity. That's Newton's First Law. Since this world is our creation, we do not have to stick to these laws, if we wish. This is the place where collisions are tested for. We will call the transform which moves the object into its final position in the world frame the object-to-world transform. It will consist of some combination of rotation and translation.

3.2.3 The View Frame

Everyone in the real world has a different view of it, and the same thing applies to the world we are creating inside the computer. The only difference is that there is only one screen and therefore only one viewer. The view of the world depends on where the observer is standing and looking.

The view of the world seen by the observer is most easily represented by the view frame. This is a set of x, y, and z-axes which follow the gaze of the observer. Usually the z-axis points forward and in our convention the x-axis points vertically up. In this picture, an object which is straight ahead at a distance of 100m will have the coordinates (0,0,100) in the view frame and if the observer rotates to the left by 90 degrees it will have view frame coordinates (0,100,0). In general the view frame's position in the world frame will be changing continuously. In a flight simulator, for example, the view frame is the view from the cockpit.

It might appear at first sight that there is an unnecessary duplication of points of view in all these frames of reference. However they define a natural hierarchy within which the overall picture can be constructed to make it easy to take account of the relative motions of the observer and graphics primitives (objects).

One thing in particular is worth noting. Rotating the view frame to the left or moving the scene to the right results in the same relative motion and gives the same picture on the screen. This suggests that there is a simple connection between two motions. In the language of mathematics, one is said to be the the inverse of the other. We will return to this again when we look at the rotations in detail. This point is examined in detail in Appendix 7.

3.2.4 The Screen

This is the logical screen, the block of RAM on which pictures are drawn before being displayed. It is mapped out following the way RAM is allocated to the screen, which in turn depends on the screen resolution, as described in Chapter 2. This results in the origin (the point with screen coordinates (0,0)) being right at the top left hand corner of the screen. To get from the view frame to the screen we must make a 'projection' onto a plane, called the view plane, of the objects which we wish to display. This is called a perspective transform and must preserve the ordering in space, so that objects which are farther away look smaller. It is done by tracing "rays" from objects to the view point, which is the location of the

observer's eye. The intersection of these rays with the view plane defines the outlines as they will appear on the screen.

The transform to the screen coordinate system is almost the last stage, but not quite; the screen has limits. It may turn out that parts of the picture lie outside the screen RAM; that part of memory allocated to the screen. If no attempt is made to restrict points to appear on the visible screen then the program will attempt to plot them outside screen RAM, which could lead to a system crash. For this reason, unless it is absolutely certain that no point to be displayed will ever lie outside the screen RAM, only part of what is visible on the view plane will reach the screen. This is ''windowing''. What is not visible must be ''clipped'' away. The outline which defines the window on the display is called a view port. To express clearly the effort that has gone into producing the final image, this is sometimes also called the clip frame.

There is even a need to clip in three dimensions in the view frame itself. Objects which are a long way away from the observer should not be displayed, and no time should be wasted worrying about them. It is a consequence of having a finite drawing resolution on the screen that small objects become badly distorted. Ultimately all very distant objects will end up as single pixels and the horizon could have a cluster of dots all over it. Sets of parallel lines will ultimately converge to a single line which will then never diminish in intensity. To stop all of this it makes sense to clip out altogether objects which are more than a certain distance from the origin of the view frame.

3.3 Coordinate Systems

When we try to put all of these transforms on a mathematical basis we immediately run into a sticky and irritating problem - how to define the coordinate systems. It is standard in engineering, science and most of mathematics to work in right-handed cartesian coordinates. A right-handed and a left-handed cartesian coordinate system are both shown in figure 3.2. In keeping with this convention we will also always use a right-handed Cartesian coordinate system. However, be warned, this is not standard in the world of computer graphics. Left-handed systems abound and sometimes both conventions are used at the same time!

There is another frequently used convention within computer graphics which, if we are to stick with it, forces the orientation of the axes in the view frame. It is that the positive z axis points forward into the picture, along the direction in which the observer is looking.

Putting all this together, we have chosen to end up with the various coordinate systems shown in figure 3.1. Positive x is up and in the world frame the y-z plane defines ground level.

Coordinate systems and frames of reference are also discussed in Appendix 7.

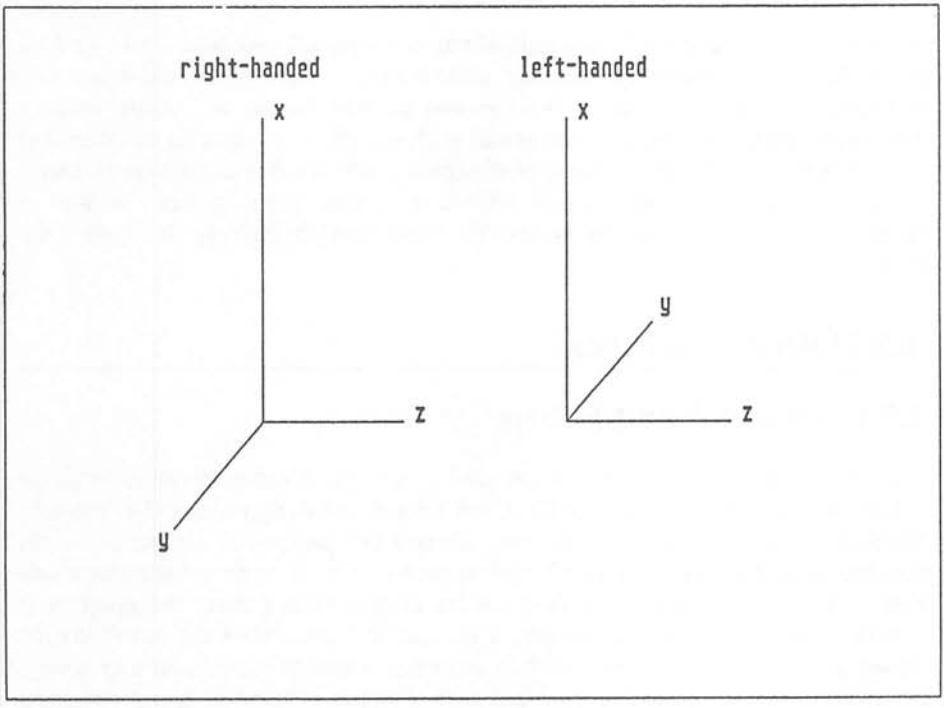

Figure 3.2 Right-handed and left-handed coordinate systems

3.4 Vectors and Matrices

For someone who loves computing but not mathematics, the introduction of matrices and vectors is not very welcome. Although it is possible to do all of the required mathematics by straightforward algebra, vectors and matrices establish an elegant and consistent framework within which to work. In addition there are properties of matrices which make them especially useful. An example is when a series of transforms take place in succession, such as when a rotation of an object about the x-axis is followed by a rotation about the y-axis. Instead of calculating the coordinates of the object twice, after each rotation, it is possible to concatenate (multiply together) the two transformation matrices and then perform the combined transform once only. This can save a lot of time when there are many points to transform.

We will discuss the various types of transforms in detail as they come up. Appendix 6 also explains matrices and vectors.

3.4.1 Vectors

Vectors are a mathematical shorthand notation which tell you how far to go in a given direction. Vectors go together with matrices. Here again there are two conventions concerning vectors. Vectors can be row vectors or column vectors. This doesn't mean very much, except that it changes how a vector looks when it is written down and the arrangement of elements inside the transformation matrices. In the teaching of engineering and science it is more usual to write vectors in column form and we will adhere to this convention exclusively throughout the book.

3.5 Data Structures

3.5.1 Variables and Labels

One of the most difficult things to get used to when first using assembly language is that there are no algebraic variables, just data stored in registers and at memory locations. You can't add x to y but you can add the contents of register *d0* to the contents of register *d1*. In a 16 bit system such as the ST even memory locations become hard to locate because they are not always known when the program is written. This is in contrast to simpler 8 bit micros where *PEEK*ing and *POKE*ing allows access to anywhere in RAM at addresses which will be fixed and always available to the program. The problem with a micro with an advanced operating system, like the ST, is that until a program is actually loaded in the machine and ready to run, its exact location will not be known. There is a way of forcing the Operating System to load the program at a particular memory location by the use of absolute code (set by the assembler directive *ORG*) but that builds inflexibility into the program and may lead to clashes with other software. That may not be a problem with a game which will tie up the computer all to itself, though it may fall victim to later modifications in the operating system.

The general philosophy is to produce programs which are insulated from all of this and come as complete self-contained packages which can be located and run anywhere in RAM. At first sight there appear to be insurmountable problems with this approach: how can you set up a table of data and later find it and how can you set up a table of addresses (jump vectors) of subroutines to execute depending on the outcome of a test? There are various solutions to these problems, some of which utilise particular addressing modes of the processor and others of which rely on the assembler, as we have already mentioned in the discussion of position independent and relocatable code in Chapter 1. The problem is solved by the extensive use of labels which are temporary substitutes for addresses which will be calculated later.

Labels play a very prominent part in any assembler program. The way they appear in the code makes them look like algebraic variables but they are not. A label is a pointer to a memory location where the current value of a variable is held, or it is a pointer to another part of the program. This is where much of the difficulty arises.

3.5.2 Lists

Finding ways of efficiently storing and accessing data has been the subject of intense study in computing. In computer graphics it is very important, particularly where speed matters. The important thing is to store data in a form such that is easy to get at for the problem in hand. It may not always be in the best form for all applications all the time, and some manipulation may be required along the way.

In vector graphics where primitives are modelled by polyhedral structures with polygonal faces, what is most important are lists of vertices (corners) and the straight line edges joining them. Figure 3.3 illustrates a house modelled in this way. There is more than one way of setting up a data list to describe this structure, but the one we will most commonly use has at its centre the list of connections which describe the surfaces uniquely: the edge list. One thing to avoid is having to repeat the actual coordinates of the vertices more than once. It is better to give each vertex a number and instead refer to this. When the x, y and z-coordinates of a vertex are required they can be drawn from the list of coordinates by the powerful indexed addressing modes of the ST, providing the position in the list is simply related to the vertex number. To make this point clear, here are the lists which are needed to draw the house. There will be other lists as well, containing other attributes such as the colour of each surface and so on, but they are not shown here. The house is not very complicated, but sufficiently so to show how long the lists might become for a really complex object.

First the number of polygons in the house as a whole must be specified. Each plane face qualifies: four walls, two sloping roofs, one floor, one door, so we have:

surface number: 8

There is only one entry here but if there were other buildings it would be a list.

Then the number of edges in each surface is given, where the entry has the same position as the number (circled) of the surface as shown in the figure:

edge numbers: 5, 4, 5, 4, 4, 4, 4, 4

After this the ordered list of vertex numbers going clockwise round the exterior face makes up the edge list. To make the data most useful to the program, the first vertex for each surface is again repeated at the end of its group to make a closed loop.

edge list: 7,8,9,2,1,7,1,2,3,4,1,4,3,10,5,6,6,5,8,7,6,5,10,9,8,5,2,9,10,3,2
1,4,6,7,1,11,12,13,14,11

Finally the actual coordinates, in whatever scale is being used, are given for x, y and z in the order of vertex numbers:

x coordinates: 0,100,100,0,100,0,0,100,150,150,0,50,50,0

y coordinates: 50,50,50,50,-50,-50,-50,-50,0,0,50,50,50,50

z coordinates: -100,-100,100,100,100,100,-100,-100,-100,100,-10,-10,10,10

These data would be used to define the house in the object frame. Following the transformation to the world frame some of the lists, the edge list, the edge numbers and the surface number would all be unchanged but the coordinates in the world frame would be different.

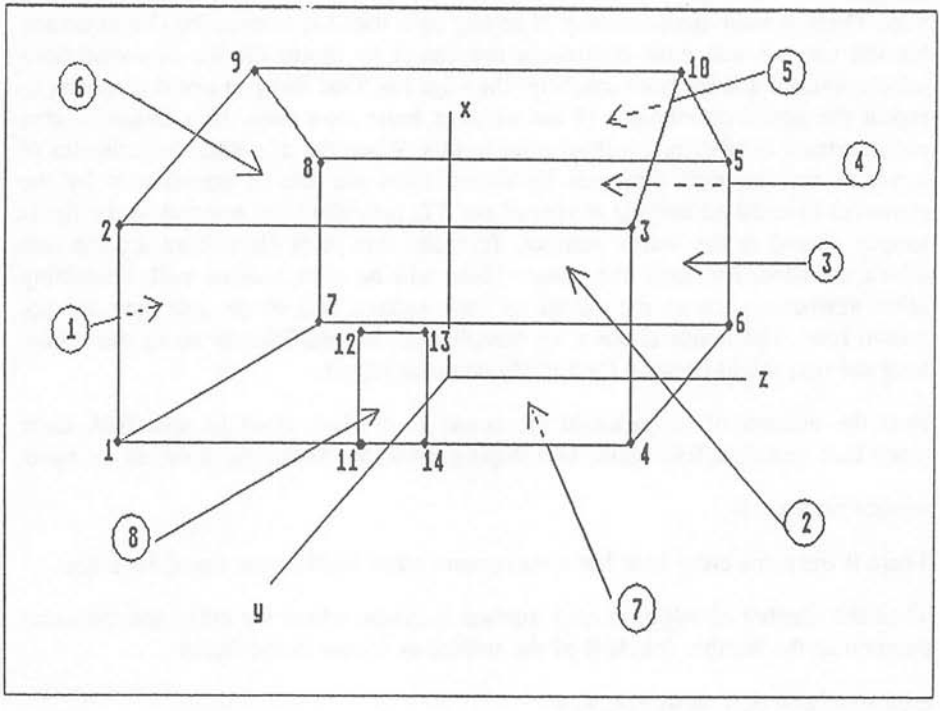

Figure 3.3 A house modelled as a polygon mesh

3.6 Summary

What should be one's attitude towards these very mathematical aspects of 3D graphics? If you are mathematically inclined, then it makes sense to try to understand what's going on in detail. This gives you the power to write your own transforms and explore some of the very interesting effects that can be produced. If you are not mathematically inclined then just regard the mathematical transforms as software "black boxes" to be "plugged in" as required. The transforms in this book are structured to allow you to do this. You only have to understand how to present data to them.

4

Fast Filling a Polygon

At the heart of our fast graphics program is the the routine which fills in a polygon. Using polyhedra as models for solid 3D objects will produce many polygonal surfaces to fill in. Because speed is of the essence, this is done in a way which fits in most easily with the way the computer works. The polygon is filled in a series of raster scans, or horizontal lines, starting at the top and progressing a line at a time to the bottom. Chapter 2 has already shown examples of fast fill routines using this method. This task naturally divides into two parts. In the first part the x coordinates of the polygon boundary are calculated and stored in a buffer (at *xbuf*) of long-words in order of increasing y. Each long-word holds in its high word the start x coordinate of the line and in its low word the end x coordinate of the line. In the second part these coordinates are passed in succession to the horizontal line drawing routine which connects them. Such an approach is called a raster scan conversion.

The first part looks, at first sight, as if it will require considerable calculation, especially if the mathematical equation of a straight line is used to find the coordinates of each (x,y) pair along it. Fortunately the solution to this problem was solved many years ago in 1962 by J.E. Bresenham. The problem at that time was to control a digital plotter which could neither multiply nor divide. Such operations are available on the 68000 but they are time consuming and we want to avoid them where possible. The great advantage of the Bresenham algorithm is that it can find all the screen coordinates of a line using only additions and subtractions. When described in algebraic terms the Bresenham algorithm looks intimidating but, like all great ideas, is really very simple. Of course some (though not all) commercial programs use algorithms which draw lines and fill polygons faster than the the Bresenham method will allow, but having understood it you can try to do better. In any case it is very elegant and very fast.

The problem facing us is to find the (x,y) coordinate pairs along the sides of a polygon so that we can use them as the start and end points for horizontal lines to do a fill. The fill of a very small area, chosen so to exaggerate the irregularity caused by the pixels, is shown in figure 4.1. Regarding the boundary as a line, we see that it looks different in different screen resolutions. At the highest resolution, the position of a pixel on the screen is specified by an integer value between 0 and 639 horizontally and between 0 and 399 vertically. With this limitation any line (unless it is either horizontal or vertical) will, under a magnifying glass, look like a staircase. This is shown in figure 4.2. In low resolution, which is the one we shall concentrate on, x has integer values between 0 and 319 and y has integer values between 0 and 199 so the staircase is easily visible. There is clearly no need for us to try to calculate the coordinates of a point to better accuracy than the screen resolution will allow, which means that integer arithmetic is quite adequate. There is no point in calculating the position of a point on the screen to 4 places of decimals because it can only be plotted to no places of decimals. The Bresenham strategy owes its success to the way it fits in with the pixel layout of the screen. Here is the way it works.

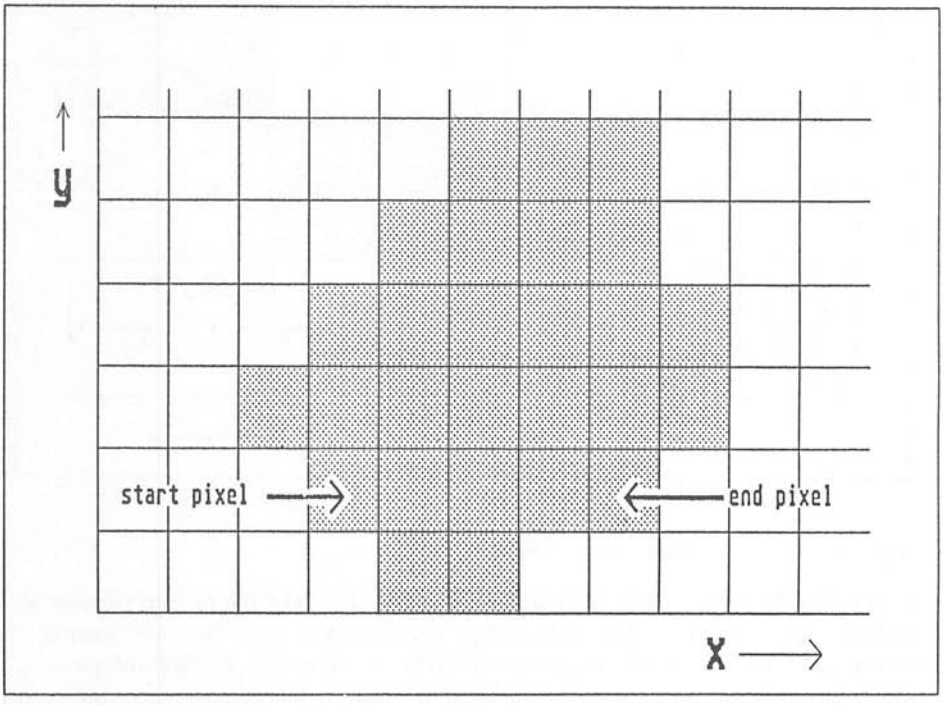

Figure 4.1 A small polygon enlarged to show pixels

4.1 Bresenham Algorithm for Drawing Lines

Let us suppose that we are plotting a line on the screen which starts at the point S(x1,y1) and ends at point T(x2,y2) as shown in figure 4.2. These points will, of course, lie precisely on the line. Now we could take a pencil and ruler and draw an ideal mathematical line between the two end points and then shade in those pixels which lie closest to the line. This is how our line will look on the screen. The result is shown in the figure where the pixels are represented by squares. We want an algorithm to do what the human brain does automatically in deciding which points to shade.

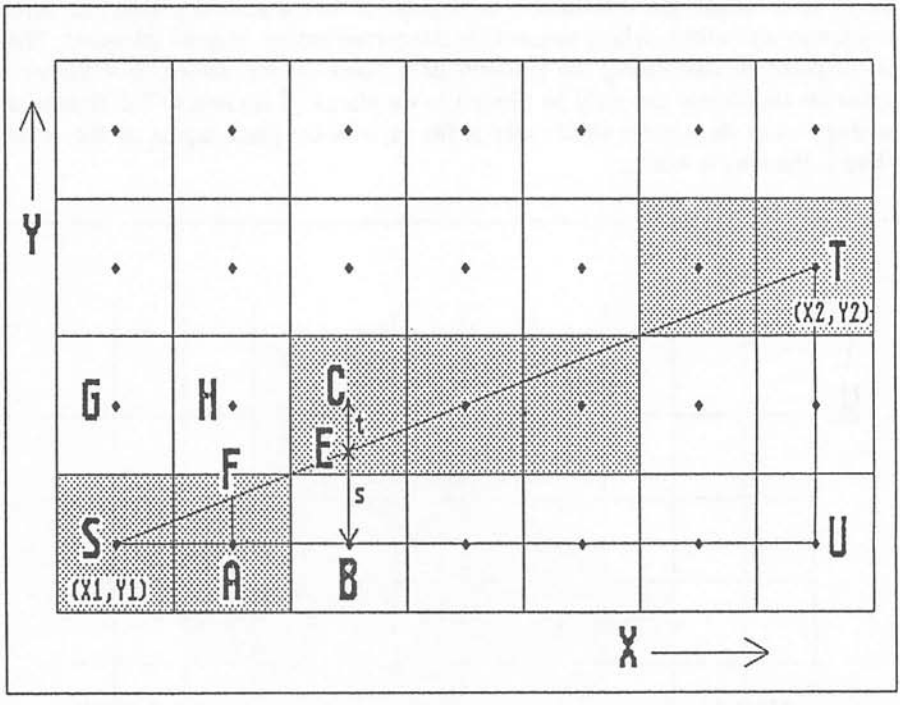

Figure 4.2 Pixel positions along a line

Here is the Bresenham algorithm which does this. To make the picture simpler we replace each pixel by a dot at its centre which makes very clear the degree to which each pixel misses the ideal line. Suppose we have just reached the point A, which didn't lie precisely on the line, and we have to choose which point to do next. The next point could be B(x+1,y) or C(x+1,y+1). It seems an obvious choice; point C because it is closer. Closer in this sense means a shorter vertical distance to the line at the point E from the centre of the pixel. We can call this the error.

On the diagram, error t is less than error s. Notice that somehow we didn't consider point H in this decision. That's because the angle of the line is less than 45°. If the angle had been greater than 45°, we would have considered the points H and C. Already it is clear that lines of slope less than 1 (angle less than 45°) are a different case from lines of slope greater than 1 (angle greater than 45°). We will come back to this later.

Well it looks like the problem is solved! Just inspect the next two points ahead, like B and C, calculate the vertical distance of each to the line and choose the shorter. In principle that's it. If the vertical distance up to the ideal line is taken as a positive error (like s) and a vertical distance down to the line is taken as a negative error (like t) then the overall quantity on which the choice is based is (s-t):

if (s-t) = D is positive, the next point is C

if (s-t) = D is negative, the next point is B.

The quantity (s-t) is called the decision variable D for obvious reasons.

Bresenham's great innovation was to spot two tricks to make this a simple operation. The first is that since only the sign of (s-t) matters, any quantity which is proportional to (s-t) will do. The second is that there is no need to re-do this calculation each time. The value of D used for the present choice can be quickly corrected to find the value of D for the next choice.

So it goes like this. The updated decision variable, D, is tested to see if it is positive or negative. If it is negative the next point to set is B. Then D is updated accordingly. If it is positive, the next point to set is C. Then D is updated accordingly. We just have to find out what these updates are and what the value of D at the very start of the line should be.

The key to answering these questions is to look at how to get from A to B or from A to C. To get from A to B do a horizontal move; to get from A to C do a horizontal followed by a vertical move. To calculate the errors associated with the individual horizontal and vertical moves it is simpler to look at point S. From this a horizontal move produces an error of AF, but a simple vertical move to G produces an error of -SG (points below the ideal line have a positive error and points above have a negative error). But SG is equal to SA, so we really only have to consider the relative lengths of the vertical and horizontal sides of the triangle SAF. But, very important, triangle SAF is similar to the overall triangle SUT and the sides are in proportion:

$$AF/SA = TU/SU = (y2-y1)/(x2-x1) = dy/dx$$

where dy is the overall distance in y and dx is the overall distance in x from the start to the end of the line.

As we have said, anything in proportion will do, so the errors could be taken as dy and -dx. A further factor of 2, which still keeps everything in proportion, will bring us into line with Bresenham's original scheme:

simple horizontal move: error = 2dy

simple vertical move: error = -2dx

For the actual moves from A to B or from A to C:

horizontal move (AB): error1 = 2dy

horizontal plus vertical move (AC): error2 = 2dy-2dx

These are the updates which must be made to the decision variable D, for the next choice.

Finally, what value of D should we start with? Everything works fine if we take the starting value D1 as the average error of error1 and error2

D1 = (error1 + error2)/2 = 2dy-dx

To summarise, here's the algorithm

1. initialize the first point to x1,y1 and the initial value of D to D1,

2. if D1 is -ve, increment x but don't increment y and make D = D + error1,

if D is +ve, increment both x and y and make D = D + error2

3. repeat step 2 until x = x2.

Now what about lines which have a slope greater than 1? The solution is very simple. To see it clearly, just draw a line with slope greater than 1 on a piece of tracing paper and clearly label the x and y axes. Now turn the tracing paper over. With the y axis horizontal and the x axis vertical, it now looks like our original line of slope less than 1 except that the x and y axes have been interchanged. Everything therefore works exactly as before if x and y are interchanged in the formulae.

4.2 Tailoring Bresenham to the Polygon Fill

The procedure we have described will certainly generate points along a line, but for our purpose we do not need them all. When considering lines of slope less then 1, points which lie on the horizontal part of the "staircase", such as S and A, all have the same y coordinate but different x coordinates. Only the x-coordinate of the first one, S, is required since the others, like A, will be filled in by the horizontal raster scan anyway. The first one in the line follows immediately the

change in sign of D . Our version of the Bresenham algorithm is modified to generate only the start and end coordinates of horizontal lines for raster scans to fill a convex polygon. It is not exactly a Bresenham algorithm in the usual sense since the coordinates it generates would, if plotted alone, produce a line full of holes along horizontals.

4.3 Example Programs

There is really only one example program here but it is split up into several parts for convenience and to emphasise their different functions. There is the control file with a name that reflects the function of the new program, together with a housekeeping file (*systm_01.s*), a file with the names of the variables (*bss_00.s*) and a file which contains the main subroutines. They are all linked together in the assembled program by means of the powerful *INCLUDE* directive. In the chapters that follow, new files of a similar kind are introduced. They bear the same name as the current ones but with a higher number. The new file contains the contents of the old file indirectly in the form of an *INCLUDE* directive. This way each new file does not waste space repeating previous code. You just have to refer back to earlier chapters to see what the earlier code does.

The main objective here is to fast fill a polygon defined only by its vertices, using the modified Bresenham routine discussed above to work out the outline. It is done in low resolution (maximum colour) and without windowing (next chapter). The polygon is shown in figure 4.3 with the coordinates of the vertices written in and the direction of ordering in the edge list indicated. For this routine the data list is actually the coordinate pairs themselves. Although the direction looks clockwise, it is in fact anticlockwise in terms of the conventional layout of an x-y graph because, in screen coordinates the origin is at the top left hand corner of the screen.

4.3.1 *polyfil2.s*

This is the main control program. It makes calls to everything else. It loads a set of coordinates of the vertices of a polygon into the variables required by the Bresenham routine. The Bresenham routine and the horizontal line drawing routine are now joined into a singe routine called *poly_fil* which is contained in the *core_00.s* file. You can change these coordinates to suit yourself but just make sure they don't go outside the boundary of the screen. As yet there is no 'clipping' to take care of this.

One new feature is the use of a a key-scan routine to terminate the program in an orderly fashion. This uses an Operating System BIOS call to see if any key at all has been pressed. If it has, another call shuts down the program and returns control to the calling program, in this case the assembler.

Another is the use of the Line A routine *$a00a* to get rid of the mouse icon directly.

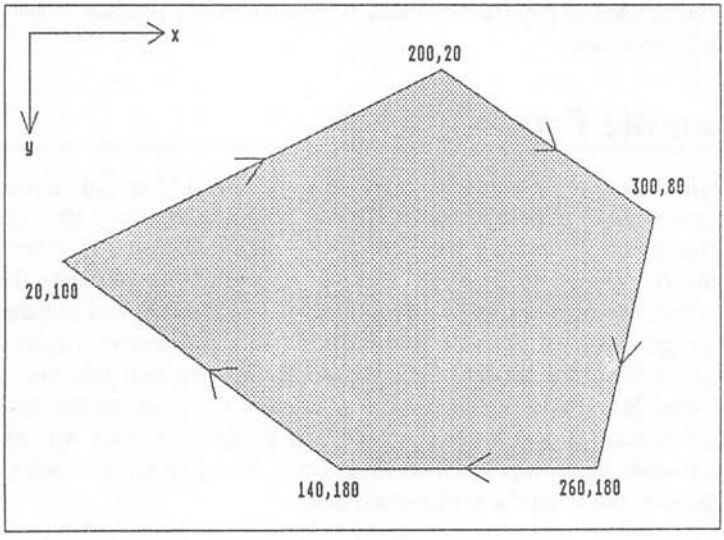

Figure 4..3 Coordinates of filled polygon

4.3.2 *core_00.s*

This is where all the work is done. It contains the Bresenham and the scan conversion programs together as a single routine *poly_fil*. The scan conversion is that used in *polfill.s*. Let's find out how the Bresenham routine works.

Pairs of vertices, defining the start and end of each polygon edge are taken from the list (x1,y1,x2,y2,...x1,y1) of coordinates, in order, going anticlockwise round the face (the direction of the arrows in figure 4.3). Notice how the first pair are repeated at the end to close the polygon. To start with, the maximum y and minimum y are saved for later use. This is a useful thing to do since this way the range of y of the polygon is found out very quickly. Then a buffer (*xbuf*) of long words is filled. Each long word contains two words which are the start and end x-coordinates of a horizontal line going from left to right across the screen. The low x is in the high word and the high x is in the low word. The position of the long word in the buffer has the value of the y-coordinate associated with the horizontal scan which is therefore an index into the buffer. Structured this way, it is clear why the maximum and minimum y of the polygon are recorded.They give the range of long words to be accessed from the buffer by the scan conversion program. With the special cases of vertical and a horizontal lines treated separately, there are really only two parts to the program: lines of slope less than 1

(angle less than 45 degrees) and lines of slope greater than 1 (angle greater than 45 degrees). These are different cases and so have to be dealt with separately.

In the case of lines of low slope, (<1), both positive and negative sloping lines are catered for with by changing the direction that x moves. For lines of positive slope (sloping forwards) x is incremented at each step and for lines of negative slope (sloping backwards) x is decremented at each step. At each step D is examined to see if it is positive or negative. If it is negative, only x is incremented and nothing is stored in the buffer at xbuf. If it is positive, both x and y are incremented and the value of x is saved in the buffer at the position dictated by y.

For lines of high slope (>1), y is incremented each time and the decision parameter tested to see if x should also be incremented. Whatever happens the value of x is saved in the buffer.

In this way the entries in the buffer fix the start and end x coordinates of the horizontal lines that fill the polygon. They are used by the routine `holine` which actually draws the lines.

4.3.3 *bss_00.s*

This is an ever increasing file of variables: quantities which are calculated during the program. This file opens space to store them.

4.3.4 *systm_01.s*

This contains three useful routines from the Operating System. Two to read the keyboard and one low level Line A routine to remove the mouse icon. The keyboard routines differ in that one only detects whether a key was pressed whilst the other records which key it was. The code for the keys are listed in Appendix 8.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                          polyfil2.s                                     *
* A program to fast fill a polygon from a set of vertex coordinates      *
* using the Bresenham algorithm to determine the outline.                *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

        SECTION TEXT
        opt     d+                  put in labels for debugging
        bra main                    here's the main program
        include systm_01.s          include the housekeeping file
        include core_00.s           and the subroutines

main    bsr     find_phys           locate the physical screen
        bsr     wrt_phys_tbl        set up the screen table
        lea     phys_tbl_y,a0       pointer to screen table
        move.l  a0,screen           pass it
        bsr     hline_lu            set up masks for filling words
        bsr     hide_mse            exterminate the mouse

* Transfer my polygon data to the program data block
        move.w  #12-1,d7            6 pairs of points for the vertices
        lea     crds_in,a0          to be moved here
        lea     my_data,a1          from here.
loop    move.w  (a1)+,(a0)+         Transfer
        dbra    d7,loop             them all.

        move.w  #5,no_in            5 sides to this polygon
        move.w  my_colour,colour        set the colour

* Generate a polygon outline in xbuf, then scan fill it.
        bsr     poly_fil            set up the buffer and fill it
* If a key is being pressed, control returns to the desktop.
        bsr     scan_keys           has a key
        tst     d0                  been pressed?
        beq     loop_again          no, execute again
        clr.w   -(sp)               gemdos call TERM - terminate
        trap    #1                  back to calling program
loop_again:
        bra     main                keep going to cover the mouse (bee)

        SECTION DATA

* A five-sided polygon (pentagon)
* Here are the vertices (first repeated at the end) in screen coords
* going anticlockwise.
my_data         dc.w    20,100,200,20,300,80,260,180,140,180,20,100
* Here is the colour - blue
my_colour       dc.w    4


        SECTION BSS
        include bss_00.s            variables calculated by the program

        END
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                          core_00.s                                    *
* Program core. Important subroutines for Chapter 4.                    *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

poly_fil:
* This fills a polygon.
* It consists of 2 parts:
* part 1 - the the x-coordinates of all boundary points are entered in xbuf
* part 2 - the holine routine fills the polygon from the values in xbuf

* PART 1. Fill the buffer.
* Regs:
* a3: pointer to crds_in - coords. list (x1,y1,x2,y2,....x1,y1)
* a2: pointer to xbuf
* d0(x1),d1(y1),d2(x2),d3(y2),d4(vertex no)/(decision ver.,
* d5(lowest y),d6(highest y)/(the increment),d7(edge counter)
* polygon vertices are ordered anticlockwise

* Initialise all variables
filxbuf:
        move.w  no_in,d7        no. edges in polygon
        beq     fil_end         quit if none to do
        lea     crds_in,a3      pointer to the coords. of vertices
        subq.w  #1,d7           the counter
        move.w  #399,d5         initial minimum y
        clr.w   d6              initial maximum y
filbuf1 lea     xbuf,a2         init. buffer pointer
        addq.w  #2,a2           point to ascending side (low word)
        move.w  (a3)+,d0        next x1
        move.w  (a3)+,d1        next y1
        move.w  (a3)+,d2        next x2
        move.w  (a3)+,d3        next y2
        subq.w  #4,a3           point back to x2
* Find the lowest and highest y values: the filled range of xbuf
        cmp.w   d5,d1           test(y1-miny)
        bge     fiibuf3         minimum y unchanged
        move.w  d1,d5           minimum y is y1
filbuf3 cmp.w   d1,d6           test(maxy-y1)
        bge     filbuf5         unchanged
        move.w  d1,d6           maximum y is y1

filbuf5 exg     d5,a5           save minimum y
        exg     d6,a6           save maximum y
        clr.w   d4              init. decision var
        moveq   #1,d6           init. increment

* All lines fall into two catagories: [slope]<1, [slope]>1.
* The only difference is whether x and y are increasing or decreasing.
* See if line is ascending (slope > 0) or descending (slope < 0).
        cmp.w   d1,d3           (y2-y1)=dy
        beq     y_limits        ignore horizontals altogether
        bgt     ascend          slope > 0
* It must be decending. Direct output to LHS of buffer. a2 must
* be reduced and we have to reverse the order of the vertices.
        exg     d0,d2           exchange x1 and x2
        exg     d1,d3           exchange y1 and y2
        subq.w  #2,a2           point to left hand buffer
ascend  sub.w   d1,d3           now dy is +ve
```

```
* Set up y1 as index to buffer
        lsl.w   #2,d1
        add.w   d1,a2
* Check the sign of the slope
        sub.w   d0,d2           (x2-x1)=dx
        beq     vertical        if it's vertical its a special case
        bgt     pos_slope       the slope is positive
* It must have a negative slope but we deal with this by making the
* increment negative
        neg.w   d6              increment is decrement
        neg.w   d2              and dx is positive
* now decide whether the slope is high (>1) or low (<1)
pos_slope:
        cmp.w   d2,d3           test(dy-dx)
        bgt     hislope         slope is >1
* The slope is less than 1 so we want to increment x every time and then
* check whether to also increment y. If so this value of x must be saved.
* dx is the counter. Initial error D1=2dy-dx
* If last D -ve, then x=x=inc, don't record x, D=D+err1
* If last D +ve, then x=x+inc,y=y+inc, record this x, D=D+err2
* err1=2dy; err2=2dy-2dx
* dx in d2, dy in d3, incx in d6, x in d0
        move.w  d2,d5
        subq.w  #1,d5           dx-1 is the counter
        add.w   d3,d3           2dy=err1
        move.w  d3,d4           2dy
        neg.w   d2              -dx
        add.w   d2,d4           2dy-dx = D1
        add.w   d4,d2           2dy-2dx=err2
        move.w  d0,(a2)         save first x
inc_x   add.w   d6,d0           x=x+incx
        tst.w   d4              what is the decision?
        bmi     no_stk          don't inc y, don't record x
        add.w   #4,a2           inc y so record x; find next buffer place
        move.w  d0,(a2)         save this x
        add.w   d2,d4           update decision D=D+err2
        bra.s   next_x          next one
no_stk  add.w   d3,d4           D=D+err1
next_x  dbra    d5,inc_x        increment x again
        bra     y_limits

* The slope is >1 so change the roles of dx and dy
* This time we must increment y each time and record the value of x after
* having done so.
* Init error D1 = 2dx-dy
* If last D -ve, then y=y+inc, D=D+err1, record x
* If last D +ve, then x=x+inc, y=y+inc, D=D+err2, record x
* err1=2dx, err2=2(dx-dy)
* dx in d2, dy in d3, inc in d6, x in d0
hislope move.w  d3,d5
        subq.w  #1,d5   dy-1 is counter
        add.w   d2,d2   2dx=err1
        move.w  d2,d4   2dx
        neg.w   d3      -dy
        add.w   d3,d4   2dx-dy=D1
        add.w   d4,d3   2dx-2dy=err2
        move.w  d0,(a2) save 1st x
inc_y   addq.w  #4,a2   next place in buffer (equivalent to incrementing y)
        tst.w   d4      what is the decision?
        bmi     same_x  don't inc x
```

```
        add.w   d6,d0   inc x
        add.w   d3,d4   D=D+err2
        bra.s   next_y
same_x  add.w   d2,d4   D=D+err1
next_y  move.w  d0,(a2) save the x value
        dbra    d5,inc_y
        bra     y_limits
* the special case of a vertical line. x is constant. dy is the counter
vertical:
        move.w  d0,(a2)         save next x
        addq.w  #4,a2           next place in buffer
        dbra    d3,vertical             for all y

* Restore the y limits
y_limits:
        exg     d5,a5
        exg     d6,a6

next_line:
        dbra    d7,filbuf1      do all lines in this polygon
next_poly:

* This part ends with minimum y in d5 and maximum y in d6

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* PART 2
* set up the pointer
        lea     xbuf,a1         base address
        sub.w   d5,d6           no. lines to do-1
        move.w  d6,d7           is the counter
        beq     poly3           quit if all sides are horizontal
        move.w  d5,d3           minimum y is the start
        lsl.w   #2,d5           4*minimum y = offset into xbuf
        add.w   d5,a1           for the address to start
        move.w  colour,d4       the colour
        subq    #1,d3           reduce initial y
poly2   addq    #1,d3           next y
        move.w  (a1)+,d2        next x1
        move.w  (a1)+,d1        next x2
        sub.w   d2,d1           x2-x1
        bmi     poly4
        addq    #1,d1           N = no to do in this line
        move.l  screen,a4       where the screen table starts
        movem.l d0-d7/a0-a6,-(sp)       save the registers
        bsr     holine                  draw the line
        movem.l (sp)+,d0-d7/a0-a6       restore the registers
poly4   dbra    d7,poly2        repeat for all y values
poly3   rts

*HOLINE. A horizontal line is drawn from left to right.
* passes: x1=d2.w, y1=d3.w, N=d1.w, colour=d4.w, screen y table:a4.l
* First find the address of the word at which the line starts.
holine  lea     hln_tbl,a3      pointer to mask table
        lsl.w   #2,d3           there are y long words before the
        movea.l 0(a4,d3.w),a4   current row address in the table
        move    d2,d5           save x1
        andi    #$fff0,d5       go in steps of 8 bytes
        lsr.w   #1,d5           to point to plane #1 word
        adda.w  d5,a4           at this address
        andi    #$000f,d2       which pixel from the left?
```

```
        move    d2,d0           save it
* does the entire line lie within one word?
        subi    #16,d0
        neg     d0              are there more pixels to the word end
        cmp     d1,d0           than we have to draw?
        bmi     long_line       no, so it's a long line
* The line is entirely within one word. Get the mask and draw it.
        move    d1,d0
        bsr     draw_it
        rts                     and that's all.
* Complete the 1st word in a long line
long_line:
        sub     d0,d1           number left
        bsr     draw_it
* Now fill all the solid words.
        clr     d0
        not     d0
        move    d1,d2           save number of pixels left to do
        lsr     #4,d2           how many are whole words?
        beq     last_word       none are
* a long stretch of filled words - no need to read the table
        subq    #1,d2           this many full words but one
        move    d0,d3           which are all 1's
        not     d3              or all 0's, depending on the colour
        moveq   #4-1,d5         4 colour planes
        move    d4,d6
        subq    #2,a4
inc_plane:
        addq    #2,a4           offset for next plane
        movea.l a4,a5           save the address
        move    d2,d7           initialise the word count
        lsr.w   #1,d6           next colour bit
        bcc     clr_word
set_word:
        or.w    d0,(a5)
        adda    #8,a5           next word in this plane
        dbra    d7,set_word
        bra     new_plane
clr_word:
        and     d3,(a5)
        adda    #8,a5           next word in this plane
        dbra    d7,clr_word
new_plane:
        dbra    d5,inc_plane    for all the colour planes
        subq    #6,a5           pointer to next plane 1
        movea.l a5,a4           update pointer
* it only remains to do the last word. It will start at pixel 0
last_word:
        andi    #$f,d1          low nibble
        cmpi.w  #0,d1           any to do ?
        beq     holine_end      no - finished.
* In finding the mask,the row offset is zero this time.
        clr     d2              1st pixel at extreme left
        move    d1,d0
        bsr     draw_it
holine_end:
        rts                     completely finished
```

```
* Draw in a section of a word which starts at pixel a and ends at pixel b
draw_it
        lsl     #5,d2           the mask row offset=a*32
        move    d0,d5           plus
        subq    #1,d5           column
        lsl     #1,d5           offset of (15-b)*2 gives
        add     d5,d2           the total offset
        move.w  0(a3,d2.w),d0   to fetch the mask
        move    d0,d3           and
        not     d3              its 1's compliment
        moveq   #3,d5           4-1 colour planes
        move    d4,d6           save colour
next_plane:
        lsr     #1,d6           is this colour bit set?
        bcc     not_set         no
        or.w    d0,(a4)+        yes, also set the bits
        dbf     d5,next_plane
        rts
not_set and.w   d3,(a4)+        clear the bits
        dbf     d5,next_plane
fil_end rts                     finished
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                        systm_01.s                                     *
*                                                                       *
* Subroutines and calls to the operating system in Chapter 4.           *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        include systm_00.s
scan_keys:
* See if a key has been pressed; don't wait (BIOS call BCONSTAT).
* returns -1 in d0 if a key was pressed
        move.w  #2,-(sp)        look at the keyboard
        move.w  #1,-(sp)        was a key pressed?
        trap    #13             bios call
        addq.l  #4,sp           tidy stack
        rts

read_key:
* Read a character from the keyboard; wait for it (BIOS call BCONIN).
* returns the code in the lower byte of the upper word of d0
        move.w  #2,-(sp)        look at the keyboard
        move.w  #2,-(sp)        wait for a key press
        trap    #13             bios call
        addq.l  #4,sp           tidy stack
        rts

hide_mse:
* Exterminate the mouse
        dc.w    $a000           init. a-line
        dc.w    $a00a           hide mouse
        rts
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                        bss_00.s                                       *
* A file of variables locations used in chapter 4.                      *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

        SECTION BSS

xbuf            ds.l    400     the buffer of x word pairs
phys_screen     ds.l    1       the address of the physical screen
log_screen      ds.l    1       the address of the logical screen
phys_tbl_y      ds.l    200     pointers to the row y's
hln_tbl         ds.w    256     the masks for filling words
screen          ds.l    1       the screen address
crds_in         ds.w    100     coords. list (x1,y1,x2,y2....x1,y1)
no_in           ds.w    1       number of sides to polygon
colour          ds.w    20      list of polygon colours
```

5

Windowing

If a picture is larger than the limits of the screen then there is a problem with what happens to the excess. Unless some provision is made for this possibility, the program will attempt to write to addresses outside of the section of RAM reserved for the screen - the screen RAM. This could be the physical screen, the 32K block of memory which is currently being displayed on the monitor or, if screen buffering is being used to produce a flicker-free picture, another 32K block, the logical screen, which is currently being drawn on and will be displayed next. Whatever the arrangement, unless we are sure that everything will always lie within the screen size, some provision must be made to clip off those sections of the picture which lie outside. Confining a picture in this way is called widowing because of the obvious analogy to someone looking out of a window. The screen is a window onto the internal world of the computer. This window could be the maximum allowed on a given resolution or something smaller (one obvious way to make graphics fast is to keep the picture small so that not much has to be drawn). The freedom to vary the size of the visible image can even give rise to special effects - an aperture opening, for example. Because of the 'clipping off' of the unwanted parts of the picture that takes place, we shall call outline of this window the clip frame.

The algorithm we need is one which will handle filled polygons. It is not sufficient to just chop off vertices where they exceed the clip frame. The line left by the chop must become an additional edge to close the polygon. Once again an elegant solution to this problem was found many years ago by Sutherland and Hodgman. Before we proceed to discuss the Sutherland-Hodgman Algorithm, it is worth mentioning that there are always shortcuts to solving problems of this kind in situations where some degree of constraint is placed on the size of graphics primitives. The way we have saved the outline of a polygon in the buffer, *xbuf*, suggests a very fast way of clipping down to a smaller frame. To provide clipping

within smaller y limits, the values of *ymax* and *ymin* can be decreased and increased respectively. Likewise a fast scan of the high and low word sides of the buffer would immediately reveal those values which exceeded the x limits, which could then be reset to the limits themselves. To stand alone, such a strategy would require that nothing ever exceeded the maximum allowed by the buffer size (which could be larger then the screen). Even if overall clipping were done separately, such a method would allow special effects such as the picture unfolding from the centre. It is always possible to achieve fast special effects by exploiting the symmetries of data structures.

5.1 Sutherland-Hodgman Clipping Algorithm

The Sutherland-Hodgman algorithm is actually more powerful than we require; it can handle polygons of any shape. In this book, for speed, only convex (round-shaped, all external angles greater than zero) polygons are filled. The requirement to be convex is a consequence of a later constraint; the need to keep the hidden-surface-removal algorithm simple. This is something we will meet at a later stage.

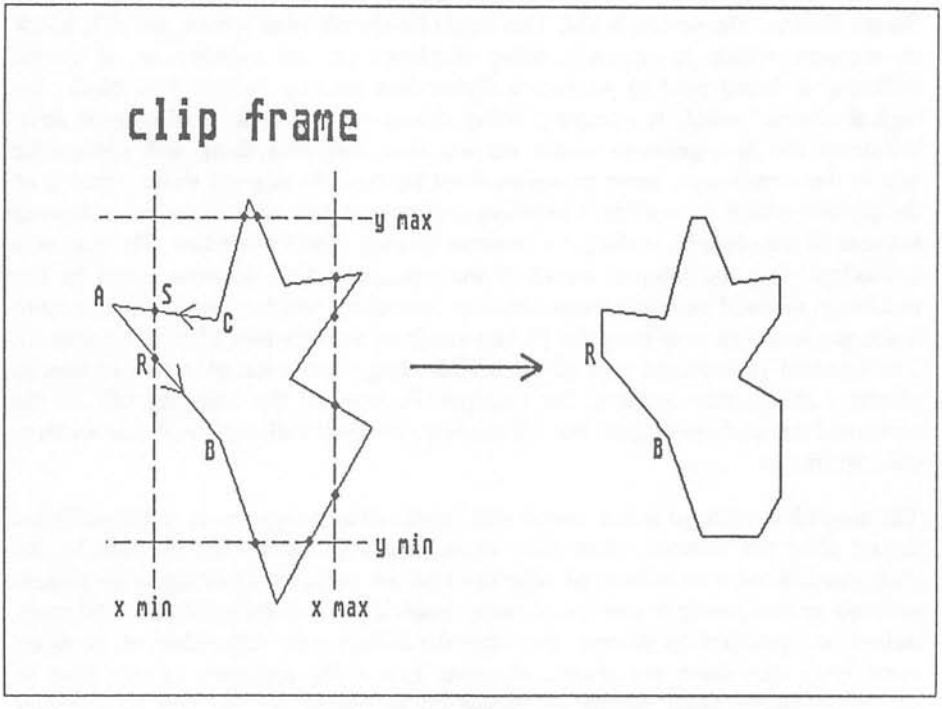

Figure 5.1 Windowing a polygon

Strictly speaking, Sutherland-Hodgman does not require polygons to be convex nor does it require the clipping frame to be a rectangle. But, for simplicity, the version given here does use a rectangular clipping frame parallel with the monitor screen. The boundaries of the clipping frame are defined by xmin, xmax, ymin and ymax and are shown for a general polygon in Figure 5.1. The Sutherland-Hodgman strategy is to find the intersections in turn of all of the edges of the polygon with each boundary. Since our boundary has four sides this means that four cycles of the polygon will be made. On each cycle some of the original edges may be lost and new ones added.

As each new vertex is examined, various actions are taken which depend on the position of it and the previous vertex. These cases are illustrated in the Figure and examined below:

1. If the next vertex is outside the frame, (A), check the position of the previous vertex, (C). If that was in, find the point of intersection, (S), of the edge joining them with the clip frame and save it. Don't save the next vertex (A).

2. If the next vertex is inside the frame, (B), check the position of the previous vertex, (A). If that was out, find the point of intersection of the edge joining them with the clip frame, (R) and save it. Also save the next vertex, (B).

This is the algorithm applied to all the vertices going round the polygon.

Once again it might appear that calculating points of intersection of sloping lines with the clip frame requires a lot of mathematical computation involving divisions and multiplications. Surprisingly this is not so. As usual in assembly language programming, where variables are not abstract algebraic symbols, but contents of memory locations or registers, it is possible to find answers using only addition and subtraction and, where it occurs, to use division and multiplication by powers of two which can quickly done by right and left shifts.

To illustrate this consider the case where the previous point was outside but the next point is inside the frame limit *xmin*. This is shown in more detail in Figure 5.2 where the two possible cases, depending on which point is closest to the limit, are examined. As part of the process to determine that B(x2,y2) lies inside and A(x1,y1) lies outside the limit, it is necessary to compare both x1 and x2 with xmin. But instead of just using the *COMPARE* instruction, the actual differences (xmin-x1) and (xmin-x2) are calculated and the sign of the result used as the basis for decision. Note that (xmin-x1) is positive and (xmin-x2) is negative. Having then decided that there is a point of intersection to determine and save, these differences are used as the stating point for calculating the point of intersection in the following way.

One of the coordinates of the point of intersection is already known; it is xmin, the limit itself; it remains to find the y value at the intercept. This is done iteratively in

the following way. The average of A and B is calculated by adding coordinates and dividing by 2. The result T1 is closer to the intercept than either A or B and we can see what side of the boundary it lies by following the sign of the average of (xmin-x1) and (xmin-x2). More important, the average of y1 and y2 will be the intercept value itself if the average of (xmin-x1) and (xmin-x2) is zero, because when this happens the two points are either evenly spaced on either side of the boundary, or coincident with it. This is the basis of the iterative algorithm used in the example program.

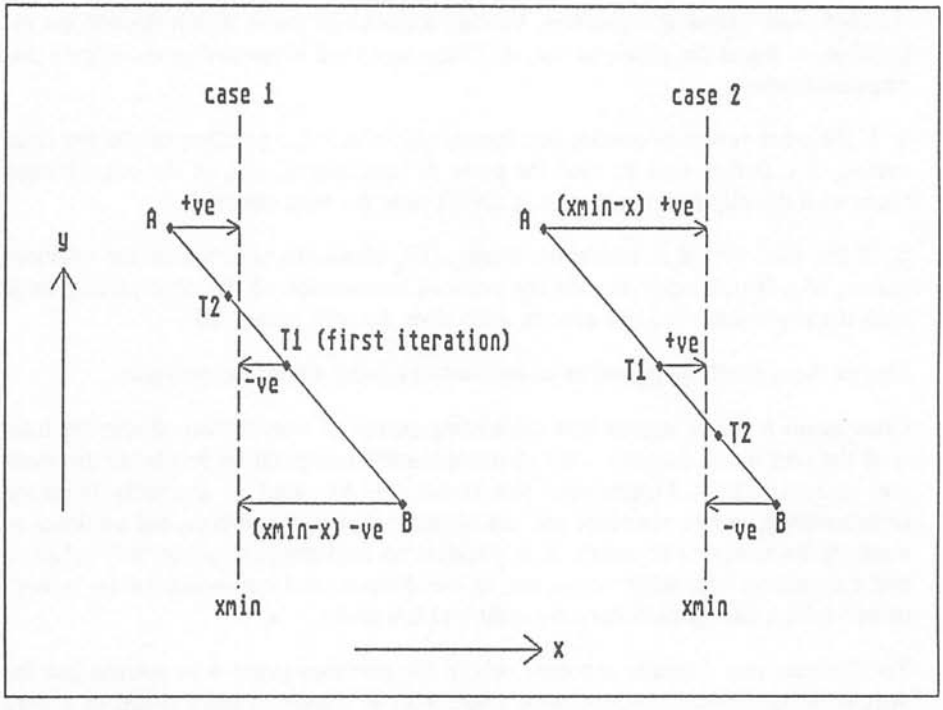

Figure 5.2 Intersection of the boundary by iteration

What happens the first time is that the average of y1 and y2 and the average of (xmin-x1) and (xmin-x2) are calculated by means of an addition and a shift right (a quick divide by two). This yields the y coordinates of the point T1. If the average of (xmin-x1) and (xmin-x2) is zero then the intercept has been found. If the x-average is negative, as at point T1 in case 1, then it lies inside the boundary and the next average must be taken between (xmin-x1) and(xmin-xT1). Likewise, the next y-average must be taken between y1 and yT1. If, on the other hand, the initial average of (xmin-x1) and (xmin-x2) is positive, as case 2, the next average must be taken between (xmin-xT1) and (xmin-x2) and the next y-average between yT1 and y2. This iterative process continues until the x-average is zero, at which point

the current y-average is the y coordinate of the point of intersection, which is then saved.

5.2 Example Program

The example program clips a polygon using a version of the Sutherland-Hodgman algorithm and then fills it. The polygon is that shown in Figure 4.3. What it looks like after the windowing is shown in Figure 5.3.

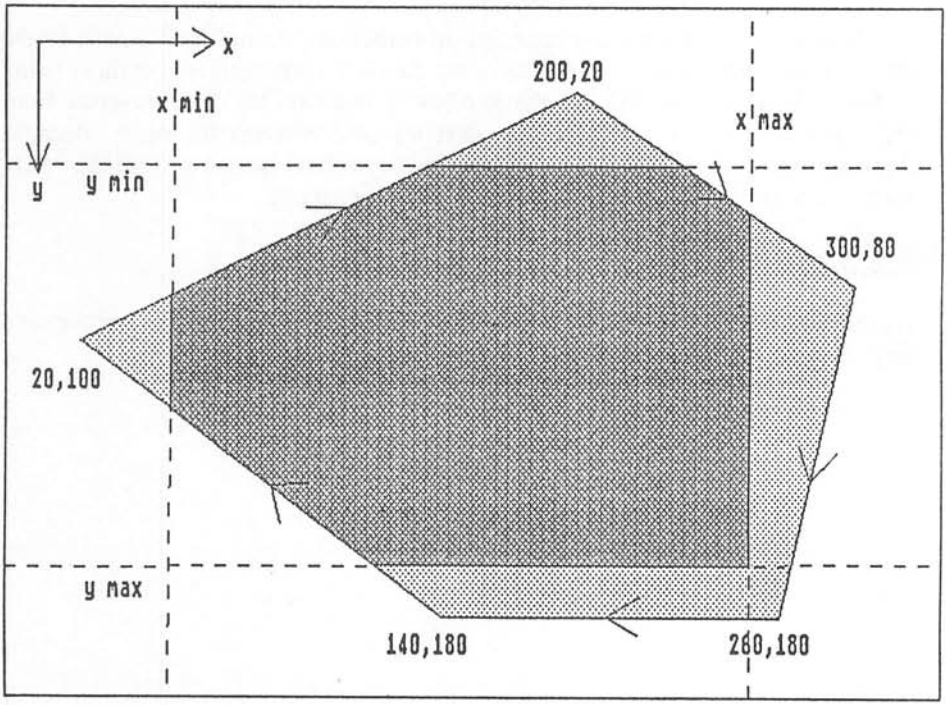

Figure 5.3 Windowed polygon

5.2.1 *clipfrme.s*

This is the control program plus the data for the polygon vertices. The coordinates in *my_data* are, as usual in the order x1,y1,x2,y2,....x1,y1, with the first coordinate repeated at the end. The clip frame limits are also given in the data and you can change them to suit yourself. The program will keep going until you press a key, when it will stop.

5.2.2 *core_01.s*

Here is where the actual clipping routine resides (together with all the other routines used so far by means of the include *core_00.s* directive at the end). Most of the work is done by the subroutine *clip*. It looks rather long but that is to try to make it more readable. Because many of its parts are very similar, it would be possible to make it shorter with inner subroutine calls, but then it would be harder to follow. It is a complicated routine but that is a consequence of the rather difficult task it does, which has been described above.

It is laid out in the order that it clips against boundaries: xmin first followed by the others. In all four complete traversals of the data are made with new vertices being added each time. The data for the vertices is input on the first traversal from *crds_in* and output to *crds_out*. The next traversal reverses the order. Because there are four traversals, the data ends up back where it started in *crds_in*, ready for the next part of the program, to follow in later chapters.

5.2.3 *bss_01.s*

As the number of variables gets larger, so does this file. So far it hasn't become so large that it has been added to with an INCLUDE directive.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                       *
*                        clipfrme.s                                     *
* A program to clip and fast fill a polygon to a window (clip frame)*
* defined by the limits stored at xmin, xmax, ymin and ymax.           *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        SECTION TEXT
        opt     d+              debugging info
        bra main
        include systm_01.s      the housekeeping file
        include core_01.s       and the subroutines

main    bsr     find_phys       locate the physical screen
        bsr     wrt_phys_tbl    where the rows start
        lea     phys_tbl_y,a0   the row look-up table pointer
        move.l  a0,screen       pass it
        bsr     hline_lu        the masks for filling words
        bsr     hide_mse        exterminate the mouse
* Set up the data
        move.w  #12-1,d7        6 pairs of points for the vertices
        lea     crds_in,a0      destination
        move.l  a0,a3           ready for drawing
        lea     my_data,a1      from here
clp_loop
        move.w  (a1)+,(a0)+     transfer
        dbra    d7,clp_loop             them all

        move.w  #5,no_in        5 sides to this polygon
        move.w  my_colour,colour        set the colour
        move.w  my_xmin,xmin    set the
        move.w  my_xmax,xmax    clip
        move.w  my_ymin,ymin    frame
        move.w  my_ymax,ymax    limits
        bsr     clip            window it
        bsr     poly_fil        fill it
loop_again:
        bsr     scan_keys       has a key been pressed?
        tst     d0
        beq     loop_again      no, try again
        clr.w   -(sp)
        trap    #1

        SECTION DATA
* A pentagon
my_data         dc.w    20,100,200,20,300,80,260,180,140,180,20,100
* which is blue
my_colour       dc.w    4
my_xmin         dc.w    50
my_xmax         dc.w    270
my_ymin         dc.w    50
my_ymax         dc.w    150

        SECTION BSS
        include bss_01.s

        END
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                          core_01.s                                    *
* Program core for Chapter 5.                                           *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

* A version of the Sutherland-Hodgman clipping algorithm.
* It goes round the polygon clipping it against one boundary at
* a time; it goes round four times in all.
* regs:
* a0(crds_in),a1(crds_out).a2(no_out),a3((saved) crds_out)
* d1(x1),d2(y1),d3(x2),d4(y2),d5(saved x2), d6(saved y2)
* d0(current limit)
clip:
* first clip against xmin
        bsr     clip_ld1                set up pointers
        tst.w   d7                      any sides to clip?
        beq     clip_end                quit if none
* do 1st point as a special case
        move.w  (a0)+,d5        1st x
        move.w  (a0)+,d6        1st y
        move.w  xmin,d0         this limit
        cmp.w   d0,d5           test(x1-xmin)
        bge     xmin_save       inside limit
        bra     xmin_update     outside limit
* do successive vertices in turn
xmin_next:
        move.w  (a0)+,d3        x2
        move.w  (a0)+,d4        y2
        move    d3,d5           save x2
        move    d4,d6           save y2
* now test for position
        sub.w   d0,d3           x2-xmin
        bge     xmin_x2in       x2 is in
* x2 is outside, where is x1?
        sub.w   d0,d1           x1-xmin
        blt     xmin_update     both x2 and x1 are out
* x2 is out but x1 is in so find intersection,
* needs dx1(+ve) in d1, dx2(-ve) in d3, y1 in d2 and y2 in d4
* finds the y-intercept and save it
        bsr     y_intercept
* but because its out, don't save x2
        bra     xmin_update
xmin_x2in:
* x2 is in but where is x1?
        sub.w   d0,d1           x1-xmin
        bge     xmin_save       both x1 and x2 are in
* x2 is in but x1 is out so find intercept
* but must have the -ve one in d3, so switch
        exg     d1,d3
        exg     d2,d4
        bsr     y_intercept

xmin_save:
        move.w  d5,(a1)+        save x
        move.w  d6,(a1)+        save y
        addq.w  #1,(a2)         inc count
xmin_update:
        move    d5,d1           x1:=x2
        move    d6,d2           y1:=y2
        dbf     d7,xmin_next
```

```
* the last point must be the same as the first
        movea.l a3,a4           pointer to first x
        subq    #4,a1           point to last x
        cmpm.l  (a4)+,(a1)+     check 1st and last x and y
        beq     xmin_dec        already the same
        move.l  (a3),(a1)       move first to last
        bra     clip_xmax
xmin_dec:
        tst.w   (a2)            if count
        beq     clip_xmax       is not already zero
        subq.w  #1,(a2)         reduce it

clip_xmax:
* Now clip against xmax. Essentially the same as above except that
* the order of subtraction is reversed so that the same subroutine
* can be used to find the intercept.
        bsr     clip_ld2        set up pointers
        tst.w   d7              any to do?
        beq     clip_ymin       no
* do 1st point as a special case
        move.w  (a0)+,d5        1st x
        move.w  (a0)+,d6        1st y
        move.w  xmax,d0
        cmp.w   d5,d0           test(xmax-x1)
        bge     xmax_save       inside limit
        bra     xmax_update     outside limit
* do successive vertices in turn
xmax_next:
        move.w  (a0)+,d3        x2
        move.w  (a0)+,d4        y2
        move    d3,d5           save x2
        move    d4,d6           save y2
* now test for position
        sub.w   d0,d3
        neg.w   d3              xmax-x2
        bge     xmax_x2in       x2 is in
* x2 is outside, where is x1?
        sub.w   d0,d1
        neg.w   d1              xmax-x1
        blt     xmax_update     both x2 and x1 are out
* x2 is out but x1 is in so find intersection
* needs dx1(+ve) in d1, dx2(-ve) in d3, y1 in d2 and y2 in d4
* find the intercept and save it
        bsr     y_intercept
* but because its out, don't save x2
        bra     xmax_update
xmax_x2in:
* x2 is in but where is x1?
        sub.w   d0,d1
        neg.w   d1              xmax-x1
        bge     xmax_save       both x1 and x2 are in
* x2 is in but x1 is out so find intercept
* but must have the -ve one in d3, so switch
        exg     d1,d3
        exg     d2,d4
        bsr     y_intercept
```

```
xmax_save:
        move.w  d5,(a1)+        save x
        move.w  d6,(a1)+        save y
        addq.w  #1,(a2)         inc count
xmax_update:
        move    d5,d1           x1:=x2
        move    d6,d2           y1:=y2
        dbf     d7,xmax_next
* the last point must be the same as the first
        movea.l a3,a4           pointer to first x
        subq    #4,a1           point to last x
        cmpm.l  (a4)+,(a1)+     check 1st and last x and y
        beq     xmax_dec        already the same
        move.l  (a3),(a1)       move first to last
        bra     clip_ymin
xmax_dec:
        tst.w   (a2)            if count
        beq     clip_ymin       is not already zero
        subq.w  #1,(a2)         reduce it

clip_ymin:
* clip against ymin
        bsr     clip_ld1        set up pointers
        tst.w   d7              any to do?
        beq     clip_ymax       no
* do 1st point as a special case
        move.w  (a0)+,d5        1st x
        move.w  (a0)+,d6        1st y
        move.w  ymin,d0         this limit
        cmp.w   d0,d6           test(y1-ymin)
        bge     ymin_save       inside limit
        bra     ymin_update     outside limit
* do successive vertices in turn
ymin_next:
        move.w  (a0)+,d3        x2
        move.w  (a0)+,d4        y2
        move    d3,d5           save x2
        move    d4,d6           save x1
* now test for position
        sub.w   d0,d4           y2-xmin
        bge     ymin_y2in       y2 is in
* y2 is outside, where is y1?
        sub.w   d0,d2           y1-xmin
        blt     ymin_update     both y2 and y1 are out
* y2 is out but y1 is in so find intersection
* needs x1 in d1, x2 in d3, dy1 in d2 and dy2 in d4
* find the intercept and save it
        bsr     x_intercept
* but because its out, don't save y2
        bra     ymin_update
ymin_y2in:
* y2 is in but where is y1?
        sub.w   d0,d2           y1-ymin
        bge     ymin_save       both y1 and y2 are in
* y2 is in but y1 is out so find intercept
* but must have the -ve one in d4, so switch
        exg     d1,d3
        exg     d2,d4
        bsr     x_intercept
```

```
ymin_save:
        move.w  d5,(a1)+        save x
        move.w  d6,(a1)+        save y
        addq.w  #1,(a2)         inc no
ymin_update:
        move    d5,d1           x1:=x2
        move    d6,d2           y1:=y2
        dbf     d7,ymin_next
* the last point must be the same as the first
        movea.l a3,a4           pointer to first x
        subq    #4,a1           point to last x
        cmpm.l  (a4)+,(a1)+     check 1st and last x and y
        beq     ymin_dec        already the same
        move.l  (a3),(a1)       move first to last
        bra     clip_ymax
ymin_dec:
        tst.w   (a2)            if count
        beq     clip_ymax       is not already zero
        subq.w  #1,(a2)         reduce it

clip_ymax:
* Now clip against ymax. Essentially the same as above except
* the order of subtraction has been reversed so that the
* same subroutine can be used.
        bsr     clip_ld2        set up pointers
        tst.w   d7              any to do?
        beq     clip_end        no
* do 1st point as a special case
        move.w  (a0)+,d5        1st x
        move.w  (a0)+,d6        1st y
        move.w  ymax,d0
        cmp.w   d6,d0           test(ymax-y1)
        bge     ymax_save       inside limit
        bra     ymax_update     outside limit
* do successive vertices in turn
ymax_next:
        move.w  (a0)+,d3        x2
        move.w  (a0)+,d4        y2
        move    d3,d5           save x2
        move    d4,d6           save y2
* now test for position
        sub.w   d0,d4
        neg.w   d4              ymax-y2
        bge     ymax_y2in       y2 is in
* y2 is outside, where is y1?
        sub.w   d0,d2
        neg.w   d2              ymax-y1
        blt     ymax_update     both x2 and x1 are out
* y2 is out but y1 is in so find intersection
* needs x1 in d1, x2 in d3, dy1(+ve) in d3 and dy2(-ve) in d4
* find the intercept and save it
        bsr     x_intercept
* but because its out, don't save y2
        bra     ymax_update
ymax_y2in:
* y2 is in but where is y1
        sub.w   d0,d2
        neg.w   d2              ymax-y1
        bge     ymax_save       both y1 and y2 are in
```

```
* y2 is in but y1 is out so find intercept
* but must have the -ve one in d4, so switch
        exg     d1,d3
        exg     d2,d4
        bsr     x_intercept

ymax_save:
        move.w  d5,(a1)+        save x
        move.w  d6,(a1)+        save y
        addq.w  #1,(a2)         inc no
ymax_update:
        move    d5,d1           x1:=x2
        move    d6,d2           y1:=y2
        dbf     d7,ymax_next
* the last point must be the same as the first
        movea.l a3,a4           pointer to first x
        subq    #4,a1           point to last x
        cmpm.l  (a4)+,(a1)+     check 1st and last x and y
        beq     ymax_dec        already the same
        move.l  (a3),(a1)       move first to last
        bra     clip_end
ymax_dec:
        tst.w   (a2)            if count
        beq     clip_end        is not alredy zero
        subq.w  #1,(a2)         reduce it
clip_end:
        rts

clip_ld1:
* first set up the pointers for the first and third passes
        lea     crds_in,a0      pointer to vertex coords. before clip
        lea     crds_out,a1     and after the this clip
        move.l  a1,a3           saved
        move.w  no_in,d7        this many sides before
        lea     no_out,a2       where the number after is stored
        clr.w   no_out
        rts

clip_ld2:
* set up the pointers for the second and fourth passes
* ensures the final output is at the same place as initial input
        lea     crds_out,a0     pointer to vertex coords before clip
        lea     crds_in,a1      and after this clip
        move.l  a1,a3           saved
        move.w  no_out,d7       this many sides before
        lea     no_in,a2        where the number after is stored
        clr.w   no_in
        rts
```

```
y_intercept:
* Find the y-intercept on the clipping boundary x = k  of the
* line joining p1(x1,y1) to p2(x2,y2).
* entry:
* d1: (x1-k) - a positive number
* d3: (x2-k) - a negative number
* d2: y1, d4: y2
        tst.w   d1              point on boundary
        beq     yint_out        already saved
        tst.w   d3              point on boundary
        beq     yint_out        will be saved
        movem   d5/d6,-(sp)     save x2, y2
yint_in move.w  d2,d6           y1
        add.w   d4,d6           y1+y2
        asr.w   #1,d6           (y1+y2)/2 = <y>, a possible intercept
        move    d1,d5           dx1
        add.w   d3,d5           dx1+dx2
        asr.w   #1,d5           ()/2 = <dx>
        beq     yint_end        if <dx>/2=0, boundary reached
        bgt     yint_loop       if not loop again
        move    d5,d3           unless <dx> is -ve, and becomes new dx2
        move    d6,d4           and <y> is new y2
        bra     yint_in         and try again
yint_loop:
        move    d5,d1           <dx> is new dx1
        move    d6,d2           <y> is new y1
        bra     yint_in

yint_end:
        move.w  d0,(a1)+        store x boundary
        move.w  d6,(a1)+        and <y> as the coords of a new vertex
        addq.w  #1,(a2)         and increment the vertex count
        movem   (sp)+,d5/d6     restore regs
yint_out:
        rts

x_intercept:
* Finds the x-intercept on the clipping boundary y = k of the
* line joining p1(x1,y1) to p2(x2,y2)
* entry:
* d1: x1, d3: x2
* d2: (y1-k) - a positive number
* d4: (y2-k) - a negative number
        tst.w   d2              point on boundary
        beq     xint_out        already saved
        tst.w   d4              point on boundary
        beq     xint_out        wil be saved
        movem   d5/d6,-(sp)     save x2, y2
xint_in move    d1,d5           x1
        add.w   d3,d5           x1+x2
        asr.w   #1,d5           ()/2 = <x> a possible intercept
        move    d2,d6           dy1
        add.w   d4,d6           dy1+dy2
        asr.w   #1,d6           (dy1+dy2)/2 = <dy>
        beq     xint_end        if <dy> = 0, boundary reached
        bgt     xint_loop       if not loop again
        move    d6,d4           unless <dy> is -ve and becomes dy2
        move    d5,d3           and <x> becomes x2
        bra     xint_in         and try again
```

```
xint_loop:
        move    d5,d1           <x> is new dx1
        move    d6,d2           and <dy> is new dy1
        bra     xint_in

xint_end:
        move.w  d5,(a1)+        store intercept <x>
        move.w  d0,(a1)+        and the boundary y as new vertex coords
        addq.w  #1,(a2)         and increment the vertex count
        movem   (sp)+,d5/d6     restore regs
xint_out        rts             next vertex
* leaves with:
* a list of vertex coordinates at coords_in
* the number of polygon sides at no_in.

        include core_00.s       add on the previous core
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*               bss_01.s                                                *
* A file of variables used in chapter 5.                                *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

        SECTION BSS

* System variables
xbuf            ds.l    400     the buffer of x word pairs
phys_screen     ds.l    1       the address of the physical screen
log_screen      ds.l    1       the address of the logical screen
phys_tbl_y      ds.l    200     pointers to the row y's
hln_tbl         ds.w    256     the masks for filling words
screen          ds.l    1       address of current screen
* Polygon atributes
crds_in         ds.w    100     input coords. list (x1,y1,x2,y2....x1,y1)
crds_out        ds.w    100     output ditto
no_in           ds.w    1       input number of sides to polygon
no_out          ds.w    1       output ditto
colour          ds.w    20      list of polygon colours
xmax            ds.w    1       clip frame limit
xmin            ds.w    1       ditto
ymin            ds.w    1       ditto
ymax            ds.w    1       ditto
```

6

Getting Things Into Perspective

It is a curious thing that distant objects look smaller than ones which are close. They aren't smaller, but they do subtend a smaller angle at the eye. For any scene to look real therefore, the size of primitives must diminish as they recede into the distance. All of this is done by the eye and the brain. Simulating the same effect on the computer screen is what the perspective transform is all about.

You don't really need to understand much maths to use the transforms in this book. The maths and the transforms have all been worked out; you only have to understand how to feed data to them. The perspective transform is just such an example. However, to understand and use transforms fully requires some understanding of maths and matrices. We will introduce these as the need arises. The Appendices also contain information on these topics

6.1 The Perspective Transform

The perspective transform is a set of mathematical operations which project an image of an object from the world reference frame onto the screen. This has a similarity to the way in which a shadow is formed, except that in that case the shadow falls behind the object and is larger, whereas in the perspective projection it is between the viewpoint and screen and smaller. This is shown in Figure 6.1.

One aspect that crops up repeatedly in transforms and matrices is the use of homogeneous coordinates. Yet it is possible to avoid using them altogether and in many cases it is an inconvenience to use them at all. What do they mean? Do they matter? In this chapter we find out about homogeneous coordinates and how to use them in the perspective transform which is done using matrix multiplication just to

illustrate the method. At the same time it will be clear how to do the transform without using matrix multiplication at all. It just turns out that the perspective transform is a good opportunity to try it out.

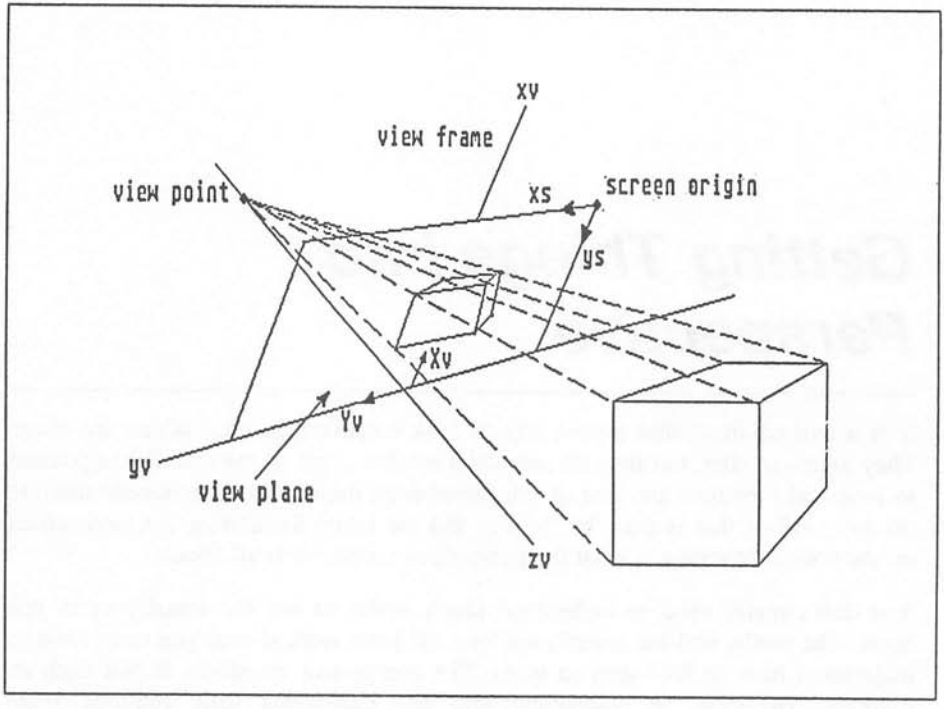

Figure 6.1 Perspective projection of a cube

Figure 6.1 shows the an object, in this case a cube, defined inside the computer in the world frame and seen from the view point. The view plane lies in the xv-yv plane of the view frame and the projected image is defined by the points where the 'rays' from the view point (also called the centre of projection, at -d along the zv axis) pierce the view plane. The window is the area of the view plane which is visible on the screen. That's really all there is to it. The view point plays a very important role in this scheme and could be placed anywhere. Placing it along the -z axis makes the algebra simple and centres the projection about the view frame origin. This is a very simple type of projection; draughtsmen use many other kinds. But it works fine and the algebra associated with it is minimal.

To make life simple, take the case where the window entirely fills the monitor screen. Then the distinction between the two disappears. Let's look at how a very simple object projects onto the screen. This is shown in Figure 6.2. As part of the transform it is also necessary to adjust to the screen coordinate system, where the origin is at the top left-hand corner. There are three coordinate systems shown in

the diagram: the view frame (xv,yv,zv), the screen frame (xs,ys), and the projected coordinates (Xv,Yv). This projected coordinate system is an intermediate one, introduced for convenience and centred at the view frame origin.

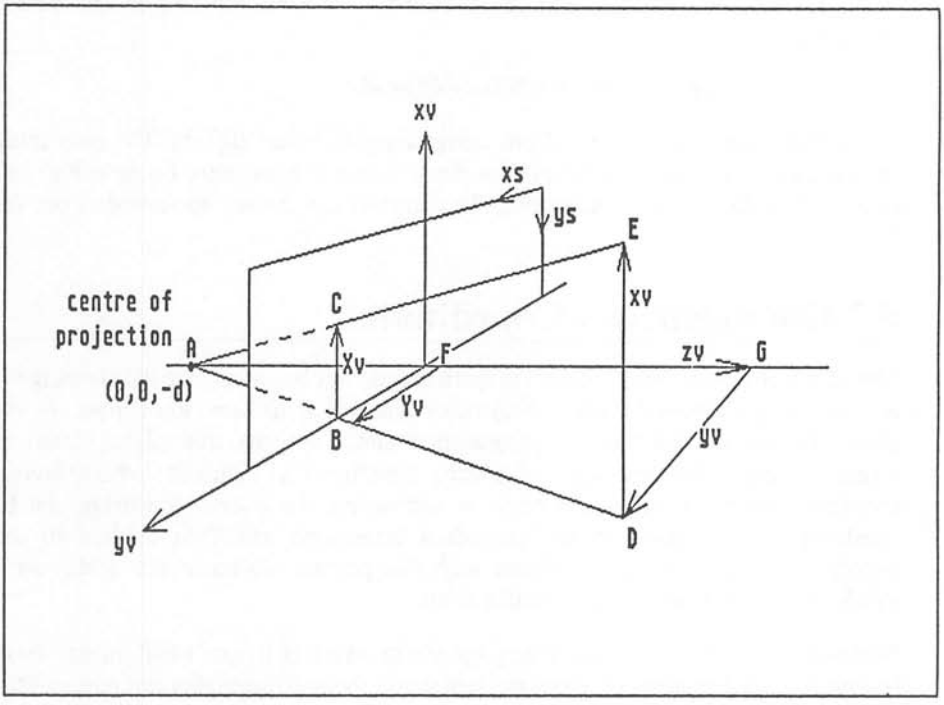

Figure 6.2 Perspective projection of a line

From the similar triangles ABC and ADE and the similar triangles ABF and ADG we get the results:

$$Xv/xv = d/(zv+d) \text{ and } Yv/yv = d/(zv+d)$$

or

$$Xv = xv.d/(zv+d) \text{ and } Yv = yv.d/(zv+d).$$

It only remains to choose where to centre the projection on the visible screen. If it is to be centred half-way across at the bottom then in screen coordinates, then

$$xs = Yv+Wx/2 \text{ and } ys = Wy-Xv$$

where Wx and Wy are the width and height of the screen in the current resolution.

In low resolution Wx=320 and Wy=200. In what follows we shall only consider low resolution, though a conversion from one resolution to another is straightforward.

In low resolution the perspective transform becomes, for display in screen coordinates:

$$xs = 160+yv.d/(zv+d) \quad ys = 200-xv.d/(zv+d)$$

These transforms can be worked out using straightforward algebra. The only thing to look out for is that the denominator doesn't ever become zero because this will cause a 'divide by zero' exception. The program can be set up to watch out for this.

6.2 Homogeneous Coordinates

The perspective transform, above, is quite simple but has a serious disadvantage if it is to be concatenated with several other types of transform. Remember, in the jargon of matrix transforms, concatenation simply means multiplying matrices together. That is the advantage of writing transforms as matrices. Where several transforms (rotations etc.) take place in succession, the overall transform can be constructed by multiplying the individual transforms and then applied to the coordinates in one go. The problem with this perspective transform is that as it stands it cannot be written as a matrix at all.

Basically, a matrix can represent any transform which is linear, which means there is a proportional relation between the initial and the transformed coordinates. What we would like to see for the transforms between Xv,Yv and xv,yv,zv are equations like

$$Xv = a.xv + b.yv + c.zv$$

$$Yv = d.xv + e.yv + f.zv$$

where the coefficients a,b,c,d,e and f are simple numbers.

Then it could be written as a matrix product (see Appendix 6 for more information on matrices)

$$\begin{pmatrix} Xv \\ Yv \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix} \begin{pmatrix} xv \\ yv \\ zv \end{pmatrix}$$

Unfortunately the perspective transform we have derived does not have this form. What messes it up is the (zv+d) in the denominators; the coordinates themselves have to be in the numerators. Therefore as it stands our transform cannot be put into 3x3 matrix form. The perspective transform isn't the only one to suffer from this problem. Simple translations do as well. The way out of the problem is to go to homogeneous coordinates.

As far as we are concerned the use of homogeneous coordinates is just a trick to get round this problem. The trick is to introduce another dimension, temporarily, to give more ''space''. That's all this extra dimension does because in this extra dimension all vertices have the same value, 1. In homogeneous coordinates the point (xv,yv,zv) becomes (xv,yv,zv,1).

How does this help? Now the transform can be written as a product but there are penalties to pay: the matrix product will generate an extra term which must be divided into the others. Also all matrices are now bigger (4x4). Here's how it works.

First do the perspective transform in homogeneous coordinates to give an intermediate result:

$$\begin{pmatrix} d.xv \\ d.yv \\ 0 \\ zv+d \end{pmatrix} = \begin{pmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & d \end{pmatrix} \begin{pmatrix} xv \\ yv \\ zv \\ 1 \end{pmatrix}$$

Then divide by the fourth element (zv+d) to give

$$Xv = xv.d/(zv+d)$$

$$Yv = yv.d/(zv+d).$$

Finally translate to the screen centre (this translation can also be done as a matrix multiplication in homogeneous coordinates but that would be making work for the sake of it):

$$xs = 160 + yv.d/(zv+d)$$

$$ys = 200 - xv.d/(zv+d).$$

The perspective matrix has zeros for most of its elements and so many of the multiplications are a waste of time. In the program at the end of this section which illustrates the transform, we have used the homogeneous form. It serves as a useful

introduction to matrix multiplication in assembly language and allows us to try a few little-used assembler instructions.

6.3 Example program

The example program shows a view of a plane with the letters "ST" (an ST monolith) sloping forwards in the world frame. When the perspective transform is done (together with windowing and everything else) it appears on the screen like

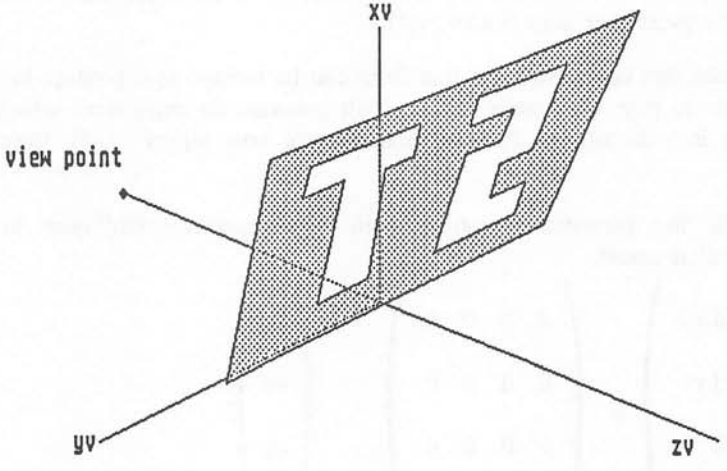

Figure 6.3 ST monolith

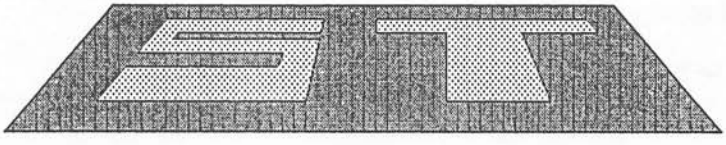

Figure 6.4 Screen picture of ST monolith

the opening logo in a movie, where the words diminish into the distance. Figure 6.3 shows how the plane is set up in the view frame. Figure 6.4 shows how it looks on the screen.

You can look at the coordinates in the data file and change them if you wish to see how it looks in different orientations. If you want, you can change the data altogether to draw something different, but first read carefully how the data is laid out. This is explained more fully below in the data file. Be careful to join up the characters and label the vertices properly.

6.3.1 *perspect.s*

This is the control program. Its function is to load up the data, draw the picture and terminate with a key press. The data are stored in the file *data_01.s*, described below.

6.3.2 *data_01.s*

This is discussed next because it contains lists of the data. Understanding how these are used is essential to understanding how the program works. Since we start off with an object drawn in 3D in the view frame, each of its vertices must be fixed by three coordinates (xv,yv,zv). The lists of these are held at *my_datax*,

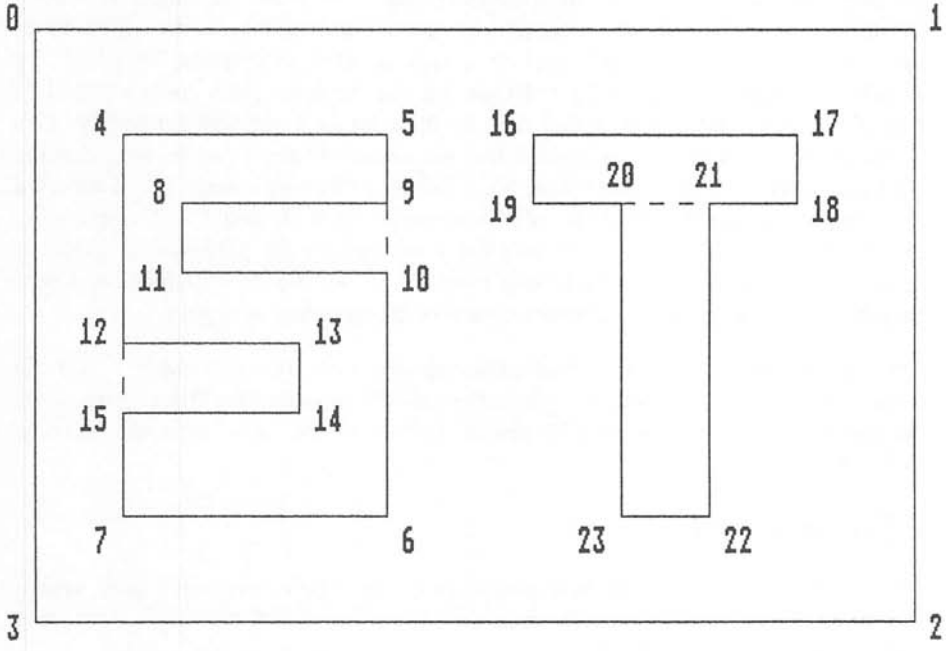

Figure 6.5 Vertex numbers of ST monolith

my_datay and *my_dataz*. There is a scheme to identify each vertex in these lists. Each vertex has a number as shown in Figure 6.5. To find its coordinates simply read in from the start counting the first coordinate as number zero. The number of vertices in each polygon is given at vectors.

More data than this is required to actually draw the picture. The connections between the vertices are specified in *my_edglst*. For each polygon there is a list of connections in this table. The overall object is split into 6 polygons, all of which lie in the same plane. The vertex connections for these, going clockwise and closing the polygon, are

polygon 0: 0,1,2,3,0

polygon 1: 4,5,6,7,4

polygon 2: 8,9,10,11,8

polygon 3: 12,13,14,15,12

polygon 4: 16,17,18,19.16

polygon 5: 20,21,22,23,20.

Arranged in this way all the information required to draw the object is readily available. To colour in the polygons a list of individual colours is held at *my_colour*. Notice that in this picture it was decided to construct the "S" by drawing an outline (polygon 1) and masking out the open parts (poly's 2 and 3) with the background colour, rather than by drawing each segment separately. This is also evident from the actual colour list, *my_colour* where it can be seen that the background is red and the letters are blue. Doing it this way saves a bit of time but may lead to problems when the blue boundaries at 9-10 and 12-15 don't quite match up. To supplement these lists the total number of polygons is given at *my_npoly*. These are the data blocks must be loaded up at initialisation. Other variables are calculated by the various parts of the program as it goes.

You can change these lists to draw anything you wish. Just remember it is a 3D object in the view frame and coordinates are easiest to determine from views along the different axes. It must also be placed in front of the view plane as shown in Figure 6.3.

6.3.3 *data_02.s*

The 4x4 matrix for the perspective transform is stored here, a row at a time, with a viewpoint at -100 on the view frame z axis. It isn't included with *data_01.s* since that file will only be used once

If you can't follow the matrix multiplication used in the transform, don't worry. Just think of the transform as a piece of 'machinery' to perform a function. If you want to alter the angle of view, change each of the numbers 100 to the new position of the view point. Remember 100 here is the distance of the view point along the negative zv axis.

6.3.4 *bss_02.s*

This contains a list of the variables used by the programs. Data is loaded into the variables blocks from the data file *data_01.s* by the control program. What goes where is clear from the control program. It consists of the lists of the x, y, and z coordinates of the vertices in the view frame, and other attributes as described in the previous sections.

6.3.5 *core_02.s*

This has two parts: the perspective transform, and *polydraw* which takes care of clipping and the actual drawing.

The perspective transform is done by matrix multiplication in homogeneous coordinates. It could be done by direct algebra but it is done this way to illustrate the use of homogeneous coordinates and matrix multiplication in a very compact way. Also it utilises a useful but little-used assembler instruction, *LINK*. When invoked, this causes the processor to open a space on the stack, called a frame, where data can be stored without interfering with the main stack. The pointer to the frame, one of the address registers, is declared in the *LINK* instruction together with the space required. The processor takes care of adjusting the regular stack pointer clear of the frame. In the present case it's where the intermediate perspective calculations are stored. When finished with, the frame is closed by means of the *UNLK* instruction and the tidying up of the stack pointer is taken care of by the processor.

The perspective transform calculates the projections of the vertices on the view plane and stores them in two lists: *scoordsx* and *scoordsy*.

Polydraw is the final part. It contains all the previous subroutines necessary to complete the drawing. It also contains at the start a test for the visibility of each polygon. This is in anticipation of things to come. The test is to look for a negative colour number. Such a value would have been set earlier if the polygon was found to be facing away from the view point.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                       *
*                          perspect.s                                   *
* A perspective view of an ST monolith                                  *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

        SECTION TEXT
        opt     d+
        bra     main
        include systm_01.s      housekeeping file
        include core_02.s       core subroutines

main    bsr     find_phys       set
        bsr     wrt_phys_tbl    up
        lea     phys_tbl_y,a0   screen
        move.l  a0,screen       routines
        bsr     hline_lu
        bsr     hide_mse
* Transfer the data. First the edge numbers and colours
        move.w  my_npoly,d7     no. of polygons?
        beq     term            if none, quit
        move.w  d7,npoly        or becomes
        subq.w  #1,d7           the counter
        move.w  d7,d0           save it
        lea     my_nedges,a0    source
        lea     snedges,a1      destination
        lea     my_colour,a2    source
        lea     col_lst,a3      destination
loop0   move.w  (a0)+,(a1)+     transfer edge nos.
        move.w  (a2)+,(a3)+     transfer colours
        dbra    d0,loop0

* Second the edge list and coordinates
        move.w  d7,d0           restore count
        lea     my_nedges,a6
        clr     d1
        clr     d2
loop1   add.w   (a6)+,d1
        add.w   d1,d2
        addq    #1,d2           last one repeated each time
        dbra    d0,loop1        = total no. of vertices

        subq    #1,d2           the counter
        lea     my_edglst,a0    source
        lea     sedglst,a1      destination
loop2   move.w  (a0)+,(a1)+     pass it
        dbra    d2,loop2
        move.w  d1,vncoords
        subq    #1,d1           the counter
        lea     vcoordsx,a1
        lea     my_datax,a0
        lea     vcoordsy,a3
        lea     my_datay,a2
        lea     vcoordsz,a5
        lea     my_dataz,a4
loop3   move.w  (a0)+,(a1)+
        move.w  (a2)+,(a3)+
        move.w  (a4)+,(a5)+
        dbra    d1,loop3
```

```
* The clip frame boundaries
        move.w  my_xmin,xmin    ready
        move.w  my_xmax,xmax    for
        move.w  my_ymin,ymin    clipping
        move.w  my_ymax,ymax

* Calculate the perspective view and draw it
        bsr     perspective
        bsr     polydraw

* Test for a key press to finish
loop4   bsr     scan_keys       test for a key press to quit
        tst     d0
        bne     term

        bra     loop4           keep testing
term    clr.w   -(sp)           terminate - return to calling program
        trap    #1

        SECTION DATA
        include data_01.s
        include data_02.s
        SECTION BSS
        include bss_02.s

        END
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                          core_02.s                                    *
* Program core for Chapter 6                                            *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

*                          perspective
* A subroutine which uses the perspective transform matrix at
* persmatx to transform a set of viewframe coords at vcoordsx,
* vcoordsy and vcoordsz into screen coords at pcoordsx and pcoordsy
* by matrix mutiplication.
* Regs:
*       a0: pointer to view frame x-coords list vcoordsx
*       a1:     ditto                      y    vcoordsy
*       a2:     ditto                      z    vcoordsz
*       a4: pointer to screen x-coords list     scoordsx
*       a5: pointer to screen y-coords list     scoordsy
* Just to be tricky we use the link instruction to open a frame on the
* stack to temporarily store the results of the calculation.

perspective:
        move.w  vncoords,d7     any points to do?
        beq     prs_end         if none, quit
        subq.w  #1,d7           otherwise this is the count
        lea     vcoordsx,a0     the
        lea     vcoordsy,a1     source
        lea     vcoordsz,a2     coords.
        lea     scoordsx,a4     the
        lea     scoordsy,a5     destination.
        link    a6,#-32         open a frame with space for 16 words
prs_crd:
* set up the perspective matrix pointer to transform the next vertex
        moveq   #3,d6           4 rows in the transform matrix M
        lea     persmatx,a3     init matrix pointer
prs_elmnt:
* calculate the next column vector element i
        move.w  (a0),d0         next view frame coord xv
        move.w  (a1),d1         next                  yv
        move.w  (a2),d2         next                  zv
        muls    (a3)+,d0        the matrix products     xv*Mi1
        muls    (a3)+,d1                                yv*Mi2
        muls    (a3)+,d2                                zv*Mi3
        add.l   d1,d0           a long word product
        add.l   d2,d0

        moveq.w #1,d1           and the extra homogeneous term
        muls    (a3)+,d1
        add.l   d1,d0           the new element

        move.l  d0,-(a6)        save it

        dbf     d6,prs_elmnt    repeat for 4 elements

        move.l  (a6)+,d3        restore 4th
        bne     prs_ok
        addq    #1,d3           avoid divide by zero
prs_ok  addq.l  #4,a6           point to 2nd
        move.l  (a6)+,d4        restore 2nd
        divs    d3,d4           next Yv=yv/(zv/d+1)result in lw
        add.w   #160,d4         centre at bottom, middle of screen
        move.w  d4,(a4)+        becomes next xs
```

```
        move.l  (a6)+,d4        restore 1st
        divs    d3,d4           next Xv=xv/(zv/d+1)
        sub.w   #199,d4         Xv-199
        neg.w   d4              199-Xv=next
        move.w  d4,(a5)+        ys

        addq.l  #2,a0           point to next xv
        addq.l  #2,a1                         yv
        addq.l  #2,a2                         zv

        dbf     d7,prs_crd      repeat for all coords
        unlk    a6              close frame
prs_end rts                     and quit


polydraw:
* This draws the visible surfaces of a polyhedron.
* It follows the perspective transform and first converts coords
* from the form of two arrays accessed from an edge list to the
* actual sequence of coord. pairs (x1,y1,x2,y2...x1,y1) needed for
* windowing and all that follows.
* Input: scoordsx pointer to list of x coords
*        scoordsy   "              y
*        sedglst - the list of edge connections (1,2,3,4,..1)
*        snedges - the number of edges in each polygon
*        npoly   - number of polygons
*        col_lst - list of colours (colour > $f means hidden)
* init. all addresses
        move.w  npoly,d7        number to do
        beq     polydraw5       there are none
        subq    #1,d7           the polygon counter
*Set up the pointers
        lea     scoordsx,a0     list (x1,x2,....xn)
        lea     scoordsy,a1     list (y1,y2,....yn)
        lea     sedglst,a2      list (1,2,3.....1)
        lea     snedges,a3      list (n1,n2,......)
        lea     col_lst,a4      list (c1,c2.....cn)
* start the loop
polydraw2:
        move.w  (a4)+,d0        colour of next polygon
        cmp.w   #$f,d0          is it visible?
        ble     polydraw3       yes
* it's hidden - update the pointers
        move.w  (a3)+,d0        no. edges in next poly
        addq.w  #1,d0           last vertex repeated
        add     d0,d0           2 bytes/word
        adda.w  d0,a2           update edge list pointer
        bra     polydraw4       go on
polydraw3:
        move.w  d0,colour       the current colour
        move.w  (a3)+,d0        no edges in next polygon
        beq     polydraw3       none to do
        move.w  d0,no_in        clip this number of edges
        lea     crds_in,a5      from this list
```

```
* set up the coords for this polygon for clip and all that follows
polydraw1:
        move.w  (a2)+,d1        next vertex no
        lsl     #1,d1           for index
        move.w  0(a0,d1.w),(a5)+        next x
        move.w  0(a1,d1.w),(a5)+        next y
        dbf     d0,polydraw1    one more coord than the count
        movem.l d7/a0-a4,-(sp)  save these
        bsr     clip            window it
        bsr     poly_fil        draw the filled polygon
        movem.l (sp)+,d7/a0-a4  restore
polydraw4:
        dbra    d7,polydraw2    for all the polygons
polydraw5:
        rts                     all done


        include core_01.s       all the previous subroutines
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                        data_01.s                                      *
*                The data file for chapter 6                            *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

* Data for current program. A large ST in perspective.

my_datax        dc.w    115,115,25,25,100,100,40,40,92,92,81,81
                dc.w    70,70,55,55,100,100,81,81,81,81,40,40
my_datay        dc.w    -100,100,100,-100,-80,-20,-20,-80,-60
                dc.w    -20,-20,-60,-80,-40,-40,-80,0,80,80
                dc.w    0,25,55,55,25
my_dataz        dc.w    120,120,0,0,100,100,20,20,90,90,75,75
                dc.w    60,60,40,40,100,100,75,75,75,75,20,20
my_edglst       dc.w    0,1,2,3,0,4,5,6,7,4,8,9,10,11,8,12,13,14,15,12
                dc.w    16,17,18,19,16,20,21,22,23,20
my_nedges       dc.w    4,4,4,4,4,4
my_npoly        dc.w    6
my_colour       dc.w    1,4,1,1,4,4
my_xmin         dc.w    0
my_xmax         dc.w    319
my_ymin         dc.w    0
my_ymax         dc.w    199
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                        data_02.s                                      *
*       A data file for chapter 6.                                      *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

* Here is the matrix for the perspective transform. The
* distance to the viewpoint along the -z axis in the viewframe
* is 100.
* The elements are given a row at a time.
persmatx:
        dc.w    100,0,0,0,0,100,0,0,0,0,0,0,0,0,1,100
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                       *
*                          bss_02.s                                     *
* Variables locations used in chapter 6.                                *
*                                                                       *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

        SECTION BSS
* System variables
xbuf            ds.l    400     the buffer of x word pairs
phys_screen     ds.l    1       the address of the physical screen
phys_tbl_y      ds.l    200     pointers to the row y's
hln_tbl         ds.w    256     the masks for filling words
screen          ds.l    1       the current screen pointer
* Polygon atributes
crds_in         ds.w    100     input coords. list (x1,y1,x2,y2....x1,y1)
crds_out        ds.w    100     output ditto
no_in           ds.w    1       input number of sides to polygon
no_out          ds.w    1       output ditto
colour          ds.w    1       current polygon colour
xmax            ds.w    1       window limit
xmin            ds.w    1       ditto
ymin            ds.w    1       ditto
ymax            ds.w    1       ditto
* Screen lists
scoordsx        ds.w    100     x coordinates
scoordsy        ds.w    100     ditto   y
sedglst         ds.w    100     edge connections
snedges         ds.w    20      no. edges in each polygon
npoly           ds.w    1       no. polygons in this polyhedron
col_lst         ds.w    20      colours of polygons
* View frame lists
vcoordsx        ds.w    100     x coords
vcoordsy        ds.w    100     y coords
vcoordsz        ds.w    100     z coords
vncoords        ds.w    1       no. of vertices
```

7

Simple Rotations

What we want to do here is rotate an object in the world frame. In our world model this is part of what happens when an object is moved from its object frame to the world frame. In addition, in general, there will be an associated translation as it is moved to its current location. As an example of simple rotations in action, the object-to-world transform is a good thing to do next. In a complex world with several different objects, each one would have different translations and rotations to bring them all together to make the world picture.

Let's take a simple world with just one object to start with. We already have a good example to work on - the monolith with the ST written on it, which was used to illustrate the perspective transform. The data is already entered and ready to go. What we would like to see is the monolith rotating in the centre of the screen. That's what we'll do next.

7.1 Geometric Transforms

Geometric transforms are those which change the coordinates of objects. Are there any other kinds? Yes, those which change frames of reference, called coordinate transforms. In mathematical language a geometric transform is the inverse of a coordinate transform (this topic is also discussed in Appendix 7). An example of the latter kind is the transform from world frame to view frame. Remember, the view frame is the set of axes attached to the observer (you) moving through the world frame. Seen from the view frame of an observer on the move, the coordinates of all objects are continuously changing. Although coordinate and geometric transforms are two sides of the same coin, the viewing transform is a bit more difficult to follow and is done later in Chapter 11.

In this section simple rotations about the x, y and z axes are presented without mathematical derivation. Turn to Appendix 7 for an additional mathematical description.

7.2 Rotations About the Principal Axes

A spinning top is a good example of an object undergoing geometric rotation about the vertical axis. As far as we are concerned here, the mathematics used to do this is just 'heavy machinery'. There is no real need to know how it is derived in order to use it. The transforms we are about to discuss are illustrated in Figure 7.1.

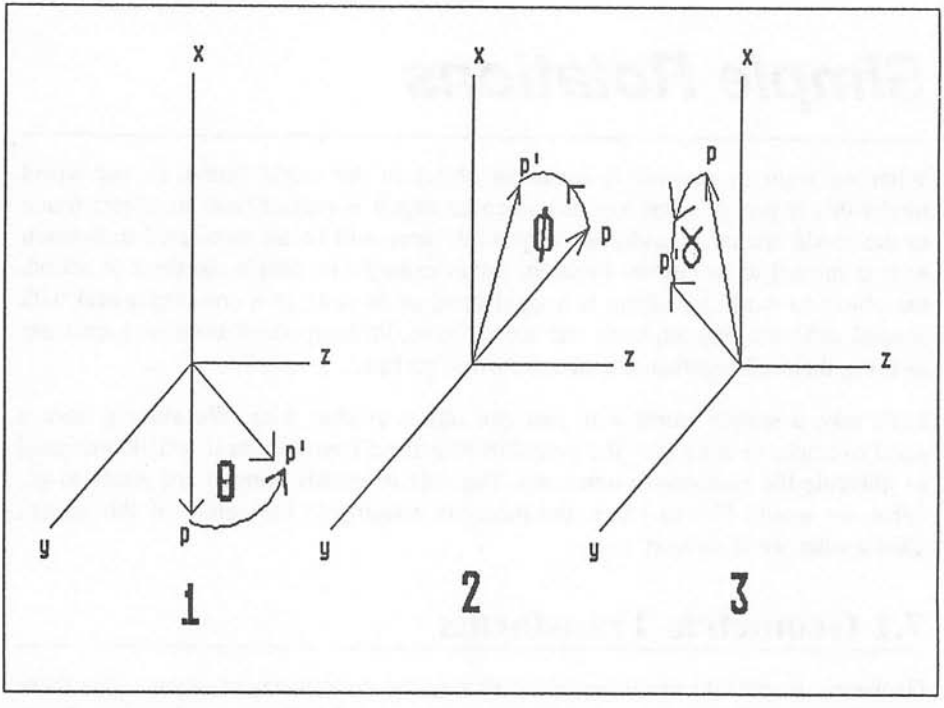

Figure 7.1 Rotations about the x, y and z axes

7.2.1 Rotation about the x-axis

This is illustrated in Figure 7.1(1) by a point P with coordinates (x,y,z) being rotated about the x-axis by an angle θ to arrive at the point P′ with coordinates (x′,y′,z′). Representing the points by vectors clearly shows the rotation. Notice how the sense of the rotation is defined. It is clockwise when looking along the positive x-axis from behind the y-z plane. In terms of the column vectors, the transform can be written as a matrix product

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

In simple algebra, with the matrix product multiplied out:

$x' = x$

$y' = y.\cos\theta - z.\sin\theta$

$z' = y.\sin\theta + z.\cos\theta$

For conciseness, the matrix is abbreviated to $R'(\theta)$ and the transform is then abbreviated to

$P' = R'(\theta).P$

7.2.2 Rotation about the y-axis

In this case the the point P is rotated about y-axis by an angle ϕ as shown in Figure 7.1(2). As before, the rotation $R'(\phi)$ is clockwise looking along the positive y-axis from behind the x-z plane. Expressed as a matrix product, the transform is

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

7.2.3 Rotation about the z axis

In Figure 7.1(3) the point P is rotated about the z-axis by an angle γ. The rotation $R'(\gamma)$ is clockwise looking along the z axis from behind the x-y plane.

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

7.2.4 Composite Rotations

When all three types of rotation are done simultaneously things become a good deal more complicated. This is because the order of rotation matters; rotating first by θ, second by ϕ and third by γ does not end up with P in the same place as with any other order. This may seem to be a surprising result. In mathematical jargon, three dimensional rotations are said to be noncommutative. To illustrate the point look at Figure 7.2.

This has two parts to it. In part 1 a vector which lies along the z axis to start with is first rotated about the x axis by 90° and then about the z axis by 90°. It ends up pointing along the x axis. In part 2 the order of rotations is reversed. Consequently

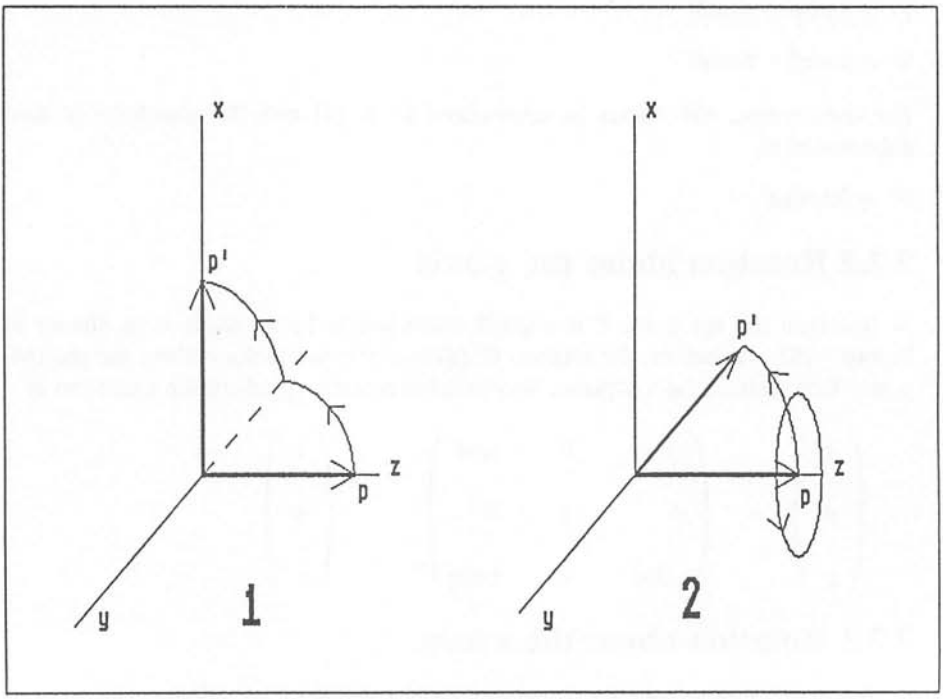

Figure 7.2 The order of the rotations matters

the first rotation does nothing and the second leaves it pointing along the -y axis. Clearly, changing the order of rotation alters the end result.

A consequence of this is that keeping count of the individual rotations θ, ϕ and γ separately provides insufficient information to get to the final position. The order of rotation must also be given. Where the individual rotations are small and frequent, such as in an object following a complex path, a different strategy must be found to keep track of the orientation.. This is discussed in Chapter 12.

For the moment this is not such a problem. Performing a simple sequence of rotations in the world frame, or as part of the object-to-world transform, may only require three rotations about the individual axes in a simple order. To have a consistent scheme, we rotate first by γ, second by ϕ and third by θ. In shorthand the overall transform when all these rotations take place in this order is:

$$P' = R'(\theta).R'(\phi).R'(\gamma).P$$

Notice how the first rotation appears next to the original point P, and later rotations appear farther to the left. This is the order of matrix multiplication with column vectors.

There is no need to perform the matrix products on the vector separately. Their product can be found beforehand to produce one resultant matrix, which can the be multiplied by the vector in one single operation. This combined (concatenated) rotation is denoted by $R'(\theta,\phi,\gamma)$.

$$R' = \begin{pmatrix} \cos\phi\cos\gamma & -\cos\phi\sin\gamma & \sin\phi \\ \sin\theta\sin\phi\cos\gamma + \cos\theta\sin\gamma & -\sin\theta\sin\phi\sin\gamma + \cos\theta\cos\gamma & -\sin\theta\cos\phi \\ -\cos\theta\sin\phi\cos\gamma + \sin\theta\sin\gamma & \cos\theta\sin\phi\sin\gamma + \sin\theta\cos\gamma & \cos\theta\cos\phi \end{pmatrix}$$

7.3 The Object-to-World Transform

This is a good transform to illustrate what we have been talking about.

The point of this transform is to move an object from its object reference frame to the world frame where it appears in the cluster of all the other objects which make up the world picture. The object-to-world transform is illustrated in Figure 7.3 for the general case of all three rotations and a translation. In this case the angles are specific to the transform and are called $o\theta$, $o\phi$ and $o\gamma$ to distinguish them from

other angles which will appear later in other transforms and the displacement is (*Oox*,*Ooy*,*Ooz*) or, written in vector notation:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = R'. \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} Oox \\ Ooy \\ Ooz \end{pmatrix}$$

Notice that the translation has not been implemented as a matrix multiplication, but has been left as a vector addition. Like the perspective transform, the translation can be converted to a matrix product in homogeneous coordinates to put it on the same footing as everything else and allow it to be included in concatenation. This is not done here because it can be incorporated simply as an addition following the rotation transform. Further information on homogeneous coordinates is given in Appendix 6.

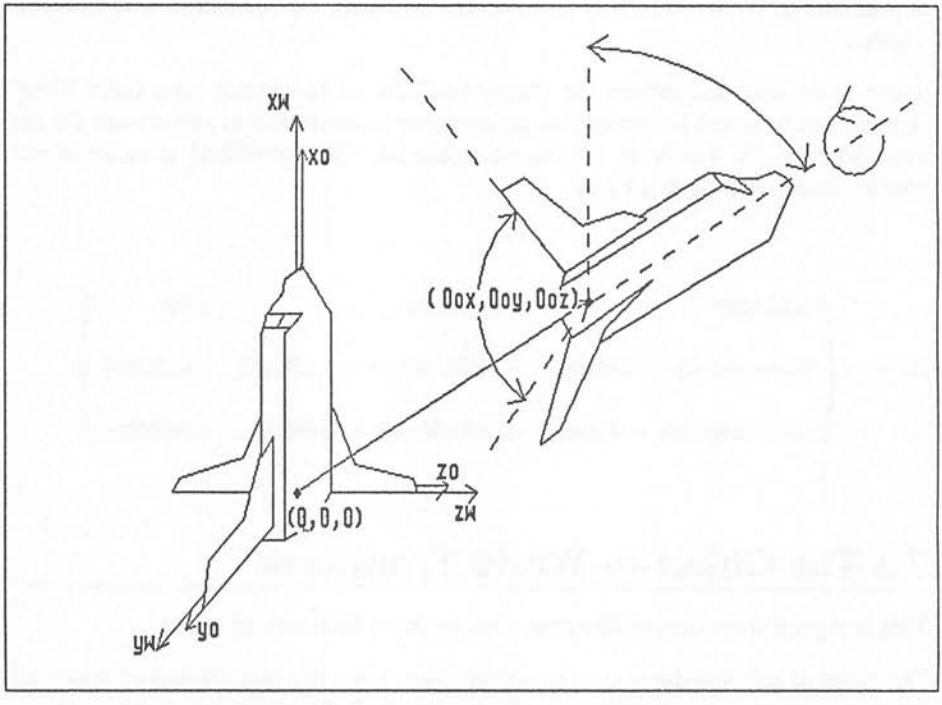

Figure 7.3 General geometric transform in the world frame

One way to think of the object frame is as a set of axes centred on the world frame origin. This is certainly a valid picture since without any rotation or translation, the object would appear at the world frame origin. The translation is essential to avoid superimposing all objects at the world frame origin. If the angles are continuously changed between frames then the object will rotate in the world frame. Since we already have the perspective transform in place from the previous chapter we can watch this happen.

7.4 Example Program

This is a program to set up the object-to-world transform and use it to show the ST monolith rotating about the z-axis of the world frame. To give a flicker-free picture, screen buffering is used. Also the sines and cosines of angles must be calculated for the rotation matrices. How these are done is discussed below in the example programs.

7.4.1 *otranw.s*

This is the main control program. This time the initialisation is more extensive because of the screen buffering and a lot of data transfer takes place. The data to draw the ST monolith is in the file *data_01.s* as before, but now it has to be transferred to the object variables list. The rotation takes place as it is transferred from the object frame to the world frame.

At the moment we can only show rotation by an angle $o\theta$ about the xw axis. This is because rotations $o\phi$ and $o\gamma$ about the other axes would try to display the rear side of the monolith. This cannot be done because of the way the polygon filling routine is set up to expect polygons in the screen frame to have an anticlockwise connected edge list . The rear side has this order reversed and in trying to cope with this the routine draws garbage. Normally the rear side of an object is not visible and would be dealt with in that way. As yet we do not have the capability to test for visibility. This is done in Chapter 9. If it were desired to show the back of the monolith it would have to be entered in the data as a separate object in a back-to-back arrangement.

Screen buffering is used to eliminate flicker effects which arise from the drawing and displaying of the picture being two separate operations, not usually in synchronisation. The problem is solved using two screens; screen1 and screen2 here. These alternately play the roles of the logical and physical screens. The logical screen is where the next frame is being drawn while the last picture is being displayed on the physical screen. The switch in the identities of these screens is made to occur when the electron beam flies back from the bottom of the screen to the top to begin drawing the next frame , called the vertical blank interrupt (or *vblank* for short). There is enough time for the switch to occur whilst this is

happening. A flag, *screenflag*, is updated each frame to keep track of the which screen is currently performing which function..

The program shows the rotation of the ST monolith about the zw axis in the world frame through the range of angles 0° to 360° in 10° steps. You can alter the angular increment between each frame and the displacement (*Oox,Ooy,Ooz*) to see what effect these have. For very large objects it is a good idea to have a small window so that only a small fraction of a large object will actually get drawn so that speed is maintained without losing the impression of size. This explains why many games have a very small window, which is the only part that needs to be re-drawn each frame, surrounded by a large static control panel which is drawn only once at the beginning.

7.4.2 *data_03.s*

The rotation transform uses the sines and cosines of the angles $o\theta$, $o\phi$ and $o\gamma$. For a program operating in Basic these would be calculated to many significant digits using a series approximation. There is no time for that here. We have to resort to the method used before hand calculators were invented – tables. The table in this file contains the sines of all the angles between 0° and 90° in 1° increments each multiplied by the factor 16384, which is 2^{14}. The reason for this is straightforward. It moves the decimal point 14 places to the left in binary and allows us to work in units of 1/16384 so that products can be determined to high accuracy. However it must be remembered that at the end of the calculation of a new coordinate the result must be divided by 16384 to restore it to its correct size. There is no point in knowing the final coordinate to greater accuracy than plus or minus 1 since this is the smallest increment which can be displayed on the screen. Also if all the trigonometric functions were not multiplied by 16384, all products would fall in the range 0 to 1 and in the approximation of binary would be approximated to one or other of these values which would then give either zero or the same result for all products. The point of choosing 2^{14} as a factor is that it can introduced or removed very quickly by 14 left or right shifts. Greater accuracy could be obtained using a larger factor, but 16384 is quite adequate for our purposes providing steps are taken to correct for errors where they occur.

For greatest speed it makes most sense to have separate tables for both sines and cosines. This is not done here mainly to illustrate how the symmetry of sine and cosines allows any value in the entire range 0° to 360° degrees to be calculated from the range 0° to 90° degrees. The time to do this is very small compared, for example, to the time taken to actually fill the polygon, but for greater speed separate tables should be used.

7.4.3 *core_03.s*

The first part of the subroutine here uses the look-up table in *data_03.s* to find the sines and cosines of the angles used in the rotation, ready for use in the transform matrix. This uses the result that the sine or cosine of any angle in the range 0° to 360° can be found from that of an equivalent angle in the range 0° to 90°. Finding this equivalent angle is what the start of the first part is all about.

In the second part, the matrix is constructed and then used to transform the object coordinates by matrix multiplication as was done in the earlier perspective transform. Although only rotations about the x axis are done in this example, the matrix can handle rotations about all three axes as described above. At the end of the rotational transform, the displacements *Oox*, *Ooy* and *Ooz* are added to place the object at the desired location in the world frame.

7.4.4 *systm_02.s*

This contains the new routines needed for screen buffering. Since two screens are now used alternately, one to draw on and one to display, the switching between them must coincide with the vertical blank interrupt to avoid flickering. In fact in both *draw1_disp2* and *draw2_disp1*, the system is made to stop and wait for this to happen by the XBIOS call number $25.

Clearing the logical screen (the one about to be written on) before it is used is a time consuming operation. It can be speeded up by clearing blocks of long words, ten at a time, using the *MOVEM* instruction. Also since there are two screens, there are two look-up tables for row addresses.

7.4.5 *bss_03.s*

New variables lists.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                otranw.s                                 *
*                            Simple Rotations                             *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

        SECTION TEXT
        opt     d+
        bra main
        include systm_02.s      housekeeping file
        include core_03.s       important subroutines

main    bsr     find_screens    find the addresses of the two screens
        bsr     wrt_scrn1_tbl   write a row address table for screen1
        bsr     wrt_scrn2_tbl   ditto                          screen2
        bsr     hline_lu
        bsr     hide_mse        exterminate the mouse
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* transfer all the data
        move.w  my_npoly,d7     no. of polygons
        beq     term            if none quit
        move.w  d7,npoly        pass it
        subq.w  #1,d7           the counter
        move.w  d7,d0           save it
        lea     my_nedges,a0    source
        lea     snedges,a1      destination
        lea     my_colour,a2    source
        lea     col_lst,a3      destination
loop0   move.w  (a0)+,(a1)+     transfer edge nos.
        move.w  (a2)+,(a3)+     transfer colours
        dbra    d0,loop0
* Calculate the number of vertices altogether
        move.w  d7,d0           restore count
        lea     my_nedges,a6
        clr     d1
        clr     d2
loop1   add.w   (a6),d1         no more than this
        add.w   (a6)+,d2        total number of vertices
        addq    #1,d2           and with last one repeated each time
        dbra    d0,loop1
* Move the edge list
        subq    #1,d2           the counter
        lea     my_edglst,a0    source
        lea     sedglst,a1      destination
loop2   move.w  (a0)+,(a1)+     pass it
        dbra    d2,loop2
* and the coords list
        move.w  d1,oncoords
        subq    #1,d1           the counter
        lea     ocoordsx,a1
        lea     my_datax,a0
        lea     ocoordsy,a3
        lea     my_datay,a2
        lea     ocoordsz,a5
        lea     my_dataz,a4
loop3   move.w  (a0)+,(a1)+
        move.w  (a2)+,(a3)+
        move.w  (a4)+,(a5)+
        dbra    d1,loop3
```

```
* and the window limits
        move.w  my_xmin,xmin    ready
        move.w  my_xmax,xmax    for
        move.w  my_ymin,ymin    clipping
        move.w  my_ymax,ymax
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* place it in the world frame
        move.w  #300,Oox        300 in the air
        move.w  #200,Ooz        200 in front
        clr.w   Ooy             dead centre
* Initialise for rotation
        clr.w   otheta          init angles
        move.w  #50,ophi        tilt it up 50 degrees
        clr.w   ogamma
        clr.w   screenflag      0=screen 1 draw, 1=screen 2 draw
        bsr     clear1          clear the screens
        bsr     clear2
* Start the rotation about the zw axis (can't rotate about the others
* or we'll see the back of it).
loop5   move.w  #360,d7         a cycle
loop4
        move.w  d7,ogamma       next angle gamma
        move.w  d7,-(sp)        save the angle
        tst.w   screenflag      screen 1 or screen2?
        beq     screen_1        draw on screen 1, display screen2
        bsr     draw2_disp1     draw on screen 2, display screen1
        bsr     clear2          but first wipe it clean
        clr.w   screenflag      and set the flag for next time
        bra     screen_2
screen_1:
        bsr     draw1_disp2     draw on 1, display 2
        bsr     clear1          but first wipe it clean
        move.w  #1,screenflag   and set the flag for next time
screen_2:
        bsr     otranw          rotational transform
* pass on the new coords
        move.w  oncoords,d7
        move.w  d7,vncoords
        subq.w  #1,d7
        lea     wcoordsx,a0
        lea     wcoordsy,a1
        lea     wcoordsz,a2
        lea     vcoordsx,a3
        lea     vcoordsy,a4
        lea     vcoordsz,a5
loop6   move.w  (a0)+,(a3)+
        move.w  (a1)+,(a4)+
        move.w  (a2)+,(a5)+
        dbra    d7,loop6
* Complete the picture
        bsr     perspective     perspective
        bsr     polydraw        finish the picture
        move.w  (sp)+,d7
* Test for terminate
        bsr     scan_keys       has a key been pressed?
        tst     d0              if so
        bne     term            back to caller
        sub.w   #10,d7          otherwise
        bgt     loop4           next angle
        bra     loop5           or repeat the cycle
```

```
term    clr.w   -(sp)           terminate and
        trap    #1              back to caller

        SECTION DATA
        include data_01.s
        include data_03.s
        SECTION BSS
        include bss_03.s

        END
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                       *
*                          data_03.s                                    *
*                     A sine look-up table                              *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

* A table of sines from 0 to 90 degrees in increments of 1 degree
* multiplied by 2^14 (16384). It can be used to find the sine or cosine
* of any angle.
sintable:
        dc.w    0,286,572,857,1143,1428,1713,1997,2280,2563,2845,3126
        dc.w    3406,3686,3964,4240,4516,4790,5063,5334,5604,5872,6138
        dc.w    6402,6664,6924,7182,7438,7692,7943,8192,8438,8682,8923
        dc.w    9162,9397,9630,9860,10087,10311,10531,10749,10963,11174
        dc.w    11381,11585,11786,11982,12176,12365,12551,12733,12911
        dc.w    13085,13255,13421,13583,13741,13894,14044,14189,14330
        dc.w    14466,14598,14726,14849,14968,15082,15191,15296,15396
        dc.w    15491,15582,15668,15749,15826,15897,15964,16026,16083
        dc.w    16135,16182,16225,16262,16294,16322,16344,16362,16374
        dc.w    16382,16384

        include data_02.s               the perspective transform
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                          core_03.s                                    *
*                  Subroutines for Chapter 7                            *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

        include core_02.s               all the previous subroutines

sincos:
* The sine and cosine of an angle are found.
* The sintable covers the positive quadrant 0 to 90 degrees
* and can be used to generate any sine or cosine in the range 0 to 360
* Entry: angle in degrees in d1
* Returns: sine in d2, cosine in d3
        lea     sintable,a5     pointer to the table base
        cmp     #360,d1         test(angle-360)
        bmi     less360         it's < 360
        sub     #360,d1         make it less than 360 degrees
less360 cmp     #270,d1         test(angle-270)
        bmi     less270         it's < 270
        bsr     over270         angle is over or equal to 270
        rts
less270 cmp     #180,d1         test(angle-180)
        bmi     less180         it's < 180
        bsr     over180         angle is over or equal to 180
        rts
less180 cmp     #90,d1          test(angle-90)
        bmi     less90          it's < 90
        bsr     over90          angle is over or equal to 90
        rts
less90  add     d1,d1           *2 for offset into table for sine
        move.w  0(a5,d1.w),d2   the sine
        subi    #180,d1         cos(angle) = sin(90-angle)
        neg     d1              offset into table for cosine
        move.w  0(a5,d1.w),d3   the cosine
        rts
over270 subi    #360,d1
        neg     d1              360 - angle
        add     d1,d1           *2 for offset
        move.w  0(a5,d1.w),d2
        neg     d2              the sine
        subi    #180,d1
        neg     d1              offset for cosine
        move.w  0(a5,d1.w),d3
        rts
over180 subi    #180,d1         angle-180
        add     d1,d1           *2 for offset
        move.w  0(a5,d1.w),d2
        neg     d2              the sine
        subi    #180,d1
        neg     d1              offset for cosine
        move.w  0(a5,d1.w),d3
        neg.w   d3              the cosine
        rts
```

```
over90  subi    #180,d1         angle-180
        neg     d1
        add     d1,d1           *2 for offset
        move.w  0(a5,d1.w),d2   the sine
        subi    #180,d1
        neg     d1              offset for cosine
        move.w  0(a5,d1.w),d3
        neg     d3              the cosine
        rts


otranw:
* This is the subroutine for transforming object coords to world coords.
* It includes rotations determined by otheta, ophi and ogamma about the
* world axes wx,wy and wz and a displacement of Oox, Ooy and Ooz relative
* to the world origin.
* PART 1.
* The matrix for the rotations is constructed.
* Convert object rotation angles to sin & cos and store for rot. matrix
        move.w  otheta,d1        theta
        bsr     sincos
        move.w  d2,stheta       store for matrix
        move.w  d3,ctheta
        move.w  ophi,d1          phi
        bsr     sincos
        move.w  d2,sphi
        move.w  d3,cphi
        move.w  ogamma,d1        gamma
        bsr     sincos
        move.w  d2,sgamma
        move.w  d3,cgamma

* Construct the transform matrix otranw remember, all elements end up *2^14
        lea     stheta,a0       sin theta
        lea     ctheta,a1       cos theta
        lea     sphi,a2         sin phi
        lea     cphi,a3         cos phi
        lea     sgamma,a4       sin gamma
        lea     cgamma,a5       cos gamma
        lea     o_wmatx,a6      the matrix
* do element OM11
        move.w  (a3),d0         cphi
        muls    (a5),d0         cphi x cgamma
        lsl.l   #2,d0
        swap    d0              /2^14
        move.w  d0,(a6)+        OM11
* do OM12
        move.w  (a3),d0         cphi
        muls    (a4),d0         cphi x sgamma
        neg.l   d0               -  "
        lsl.l   #2,d0
        swap    d0              /2^14
        move.w  d0,(a6)+        OM12
* do OM13
        move.w  (a2),(a6)+      sphi
```

```
* do OM21
        move.w  (a1),d0         ctheta
        muls    (a4),d0         ctheta x sgamma
        move.w  (a0),d1         stheta
        muls    (a2),d1         stheta x sphi
        lsl.l   #2,d1
        swap    d1
        muls    (a5),d1         stheta x sphi x cgamma
        add.l   d1,d0   stheta x sphi x cgamma + ctheta x sgamma
        lsl.l   #2,d0
        swap    d0
        move.w  d0,(a6)+
* do OM22
        move.w  (a1),d0         ctheta
        muls    (a5),d0         ctheta x cgamma
        move.w  (a0),d1         stheta
        muls    (a2),d1         stheta x sphi
        lsl.l   #2,d1
        swap    d1
        muls    (a4),d1         stheta x sphi x sgamma
        sub.l   d1,d0   ctheta x cgamma - stheta x sphi x sgamma
        lsl.l   #2,d0
        swap    d0
        move.w  d0,(a6)+
* do OM23
        move.w  (a0),d0         stheta
        muls    (a3),d0         stheta x cphi
        neg.l   d0              - "
        lsl.l   #2,d0
        swap    d0
        move.w  d0,(a6)+
* do OM31
        move.w  (a0),d0         stheta
        muls    (a4),d0         stheta x sgamma
        move.w  (a1),d1         ctheta
        muls    (a2),d1         ctheta x sphi
        lsl.l   #2,d1
        swap    d1
        muls    (a5),d1         ctheta x sphi x cgamma
        sub.l   d1,d0   stheta x sgamma - ctheta x sphi x cgamma
        lsl.l   #2,d0
        swap    d0
        move.w  d0,(a6)+
* do OM32
        move.w  (a0),d0         stheta
        muls    (a5),d0         stheta x cgamma
        move.w  (a1),d1         ctheta
        muls    (a2),d1         ctheta x sphi
        lsl.l   #2,d1
        swap    d1
        muls    (a4),d1         ctheta x sphi x sgamma
        add.l   d1,d0
        lsl.l   #2,d0
        swap    d0
        move.w  d0,(a6)+                " + stheta x cgamma
* d0 OM33
        move.w  (a1),d0         ctheta
        muls    (a3),d0         ctheta x cphi
        lsl.l   #2,d0
        swap    d0
```

```
        move.w  d0,(a6)+
* PART 2
* now the object coords are transformed to world coords
* Remember matrix elements are *2^14 and must be corrected at the end
        move.w  oncoords,d7     the number
        ext.l   d7              any to do ?
        beq     otranw3         if not quit
        subq.w  #1,d7           or this is the count

        lea     ocoordsx,a0     the
        lea     ocoordsy,a1     source
        lea     ocoordsz,a2     coords.
        lea     wcoordsx,a3     the
        lea     wcoordsy,a4     destination
        lea     wcoordsz,a5
        exg     a3,d3           save this address-shortage of regs.
        link    a6,#-6          3 words to store
otranw1:
        moveq.l #2,d6           3 rows in the matrix
        lea     o_wmatx,a3      init matx pointer
* calculate the next wx, wy and wz
otranw2:
        move.w  (a0),d0          ox
        move.w  (a1),d1          oy
        move.w  (a2),d2          oz

        muls    (a3)+,d0        ox*Mi1
        muls    (a3)+,d1        oy*Mi2
        muls    (a3)+,d2        oz*Mi3

        add.l   d1,d0
        add.l   d2,d0
        lsl.l   #2,d0
        swap    d0              /2^14
        move.w  d0,-(a6)        save it .
        dbf     d6,otranw2      repeat for 3 elements

        move.w  (a6)+,d0        off my stack
        add.w   Ooz,d0          add the displacement
        move.w  d0,(a5)+        becomes wz
        move.w  (a6)+,d0
        add.w   Ooy,d0
        move.w  d0,(a4)+        becomes wy
        exg     a3,d3           restore address wx, save matx pointr
        move.w  (a6)+,d0
        add.w   Oox,d0
        move.w  d0,(a3)+        becomes wx
        exg     a3,d3           save address wx, restore matx pointr
        addq.l  #2,a0           point to next ox
        addq.l  #2,a1                          oy
        addq.l  #2,a2                          oz

        dbf     d7,otranw1      repeat for all ocoords
        unlk    a6              close frame
otranw3 rts                     and quit
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                        systm_02.s                               *
*              Calls to the Operating System                      *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

        include systm_01.s      the earlier routines

* find the screen addresses
find_screens:
        move.w  #2,-(sp)        xbios _physbase
        trap    #14             xbios call
        addq.w  #2,sp           tidy stack
* the physical screen base address is returned in d0 and saved
        move.l  d0,screen2      as screen2
* calculate the address of the logical screen and save it
        sub.l   #$8000,d0       another 32k screen to draw on
        move.l  d0,screen1      called screen1
        rts

draw1_disp2:
* DRAW ON SCREEN 1, DISPLAY SCREEN 2 (AT VBLNK)
        move.w  #-1,-(sp)       ignore resolution
        move.l  screen2,-(sp)   display 2
        move.l  screen1,-(sp)   draw on 1
        move.w  #5,-(sp)        xbios_setscreen
        trap    #14
        add.l   #12,sp          tidy
        lea     scrn1_tbl,a0    tell the program
        move.l  a0,screen
* wait for it
        move.w  #$25,-(sp)      xbios wait for vblank
        trap    #14
        addq.l  #2,sp           trap 14
        rts

draw2_disp1:
* DRAW ON SCREEN 2, DISPLAY SCREEN 1
        move.w  #-1,-(sp)       ignore resolution
        move.l  screen1,-(sp)   display 2
        move.l  screen2,-(sp)   draw on 1
        move.w  #5,-(sp)        xbios_setscreen
        trap    #14
        add.l   #12,sp          tidy
        lea     scrn2_tbl,a0    tell the program
        move.l  a0,screen
* wait for it
        move.w  #$25,-(sp)
        trap    #14
        addq.l  #2,sp
        rts
```

```
* CLEAR SCREEN 1 (by wiping out 10 long words at a time)
clear1  move.l  screen1,a3      screen1 base
        adda.l  #32000,a3       point to top
        move.w  #799,d7
        moveq.l #0,d0
        move.l  d0,d1
        move.l  d1,d2
        move.l  d2,d3
        move.l  d3,d4
        move.l  d4,d5
        move.l  d5,d6
        movea.l d6,a0
        movea.l a0,a1
        movea.l a1,a2
clr1_1  movem.l d0-d6/a0-a2,-(a3)
        dbf     d7,clr1_1
        rts
* CLEAR SCREEN 2
clear2  move.l  screen2,a3      screen 2 base
        adda.l  #32000,a3
        move.w  #799,d7
        moveq.l #0,d0
        move.l  d0,d1
        move.l  d1,d2
        move.l  d2,d3
        move.l  d3,d4
        move.l  d4,d5
        move.l  d5,d6
        movea.l d6,a0
        movea.l a0,a1
        movea.l a1,a2
clr2_1  movem.l d0-d6/a0-a2,-(a3)
        dbf     d7,clr2_1
        rts

* Write a table of row addresses for screen1
wrt_scrn1_tbl:
        move.l  screen1,d0
        move.w  #200-1,d1
        lea     scrn1_tbl,a0
lu1loop move.l  d0,(a0)+
        add     #160,d0
        dbra    d1,lu1loop
        rts

* Write a table of row addresses for screen2
wrt_scrn2_tbl:
        move.l  screen2,d0
        move.w  #200-1,d1
        lea     scrn2_tbl,a0
lu2loop move.l  d0,(a0)+
        add     #160,d0
        dbra    d1,lu2loop
        rts
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                          bss_03.s                                     *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

        include bss_02.s
* Object frame variables
otheta          ds.w    1       the rotation of object coords about wx
ophi            ds.w    1       ditto                               wy
ogamma          ds.w    1       ditto                               wz
ocoordsx        ds.w    200     vertex x coords
ocoordsy        ds.w    200     ditto  y
ocoordsz        ds.w    200     ditto  z
oncoords        ds.w    1       number
Oox             ds.w    1       object origin x coord.in world frame
Ooy             ds.w    1       ditto          y
Ooz             ds.w    1       ditto          z
* World frame variables
wcoordsx        ds.w    200
wcoordsy        ds.w    200
wcoordsz        ds.w    200
* Variables for the o_w transform
o_wmatx         ds.w    9       the matrix elements

* General
screen1         ds.l    1       where the screen1 address is stored
screen2         ds.l    1       ditto           2
scrn1_tbl       ds.l    200     table of row addresses for 1
scrn2_tbl       ds.l    200     ditto                      2
screenflag      ds.w    1       0 to display screen1, 1 for screen2
stheta          ds.w    1       trig functions of the current angle
ctheta          ds.w    1
sphi            ds.w    1
cphi            ds.w    1
sgamma          ds.w    1
cgamma          ds.w    1
```

8

Keyboard, Joystick and Mouse

These three input devices provide a simple way of injecting data into a running program. Getting to grips with program input from these devices is straightforward but a little confusing, largely as a consequence of the number of ways of achieving similar results. There are 'standard' ways using the higher levels of the operating system, low level assembler routines which use the machine-dependent BIOS and XBIOS and 'quick and dirty' methods which spy on system variables.

Reporting what keys are pressed, how far the mouse has moved or where the joystick is pointing is all done by the intelligent keyboard (IKBD) controller. It is 'intelligent' because it is a computer in its own right and operates quite independently of the main processor. In the default setting, every time something changes on one of the input devices the IKBD passes the decoded information onto the 68000 in an interrupt. This relieves the main processor of the chore of repeatedly scanning them. The technicalities arise for us in 'grabbing' this information and using it for our own nefarious purposes. As you will see, there are some very ''sneaky'' ways of doing this.

8.1 ''Quick and Dirty''

There are always quick ways of doing things which bypass the cumbersome but thorough routines of the operating system. When it comes to identifying what keys are being pressed or how much the mouse has moved, it's important to remember that GEM is keeping track of such things all the time. The problem is to locate where the operating system stores its findings and how to interpret them. This looks like a 'needle-in-a-haystack' problem, but it isn't. We can use a talent

possessed by humans but not shared by computers to quickly solve the problem: humans are good at recognising patterns and changes in them. This can be exploited by displaying the System Variables area of RAM visually and watching how the patterns change as the input devices are operated. The only disadvantage with using System Variables this way is that, due changes in the Operating System, they may be at different locations in machines produced at different times. If you only want to write programs for yourself, this method is O.K.

To get a visual representation of the variables area, we can use the xbios routine *SETSCREEN (#5)* to make this area of RAM the physical screen. Then the contents of the memory locations are visible as set pixels on the screen. The program *ramview.s* listed at the end of the chapter does this. When the program is running you can press keys, waggle the joystick or move the mouse and see which pixels flicker. Calculating their locations in RAM can then be found fairly accurately in the following way.

Take a good ruler and measure the height of the screen - call this length Y. Measure the distance from the top of the screen to the row containing the flickering pixel - call this y. Then remembering that the overall length of the screen is 32kbytes and that it starts at the beginning of user RAM, 2049, the address you're interested in is the vicinity

address = ((y/Y)*32000) + 2049.

This is best done in high resolution.

You will see different locations for the keyboard, mouse and joystick. Try holding down a key and watching a key buffer fill up, or move the mouse around and watch the resulting frenetic activity. Apart from these input registers, it's fun to see all the other functions, particularly counters and clocks, being updated in this ''bird's eye view'' of the System.

Obviously this isn't an accurate enough result to use directly in a program, it will only be approximate. The exact location will have to be pinned down by examination of specific addresses. This is helped by the way input devices are read by the IKBD controller independent of what else is going on. The default settings of the System are such that whenever an input change is detected the result is passed on regardless. So, for example, you can use a debugger to single-step through a trivial program whilst watching memory locations at the same time as the joystick is operated. The program *key_peek.s* does this. Type in the program and assemble it to memory. Then enter the debugger and run the program by pressing the keys CTRL-Z. This will fill the buffer sufficiently to keep it going whilst you press other keys and look for a response in the registers.

The address to start with is *$e40*. The variables will be somewhere in this vicinity. Once the locations of variables are known you can read them into your programs whenever you want.

8.2 Strictly by the Book

The operating system is packed with routines to look at the keyboard but finding out what the mouse or joystick are up to is a little more difficult. We'll look at the keyboard first.

8.2.1 The Keyboard

Actually it has already been done. The routines in *systm_01.s* are all that are required and have already been used to quit programs. The routine *read_key* returns the GEM standard codes of the keys pressed. You can run this in a loop as an independent program and using the monitor to watch *d0*, find the codes for all the ST keys. The codes are also listed in Appendix 8.

8.2.2 The Joystick

To read the joystick and the mouse, subterfuge is required. It will appeal to the latent hacker present in all of us!

The IKBD has to report which joystick is being operated, whether the fire button has been pressed and in what direction the stick has moved. There is more than one piece of information here and so several bytes are passed together as a data packet. Since the IKBD works independently, it tells the main processor when a data packet is coming. The packet is then intercepted by a routine called the packet handler. The addresses of the handlers for data packets from all input devices is located in a table. This table of pointers to subroutines is called a vector table. The trick is to find the location of this table and the joystick vector within it, and then substitute our own vector, i.e. write our own joystick handler routine and place its address in the table. This way, whenever a joystick packet is sent out it will be intercepted by our own routine ready for our use. The original System vector can of course be saved and replaced when we're finished. It will not surprise you to learn that there is an XBIOS routine (KBDVBASE) which returns the base address of the vector table. The joystick vector is number six up in this table. The subroutine to make the substitution is given in the file *systm_03.s.* and is called *init_joy*. Once in place, the new handler takes the second byte in the packet and stores it in the location *joy_data* for our later use. The first byte in the packet is *$ff* for joystick 1 and *$fe* for joystick 2. This byte is of no further interest to us.

The byte which is saved contains in its lowest nibble the number signifying the direction in which the stick has been moved. This is : up - 1, down - 2, left - 4, right - 8, up-left - 5, down-left - 6, up-right - 9, down-right - 10.

There is a further complication to the business which we can avoid by setting the system up properly. It arises from the fact that at any given time both the joystick and the mouse are active and cause data packets to be generated. If no attempt is made to inspect the header byte then data from the mouse can be confused with that from the joystick. We have made no attempt to do this in our handler but the problem is simply solved by turning off the mouse altogether. To do so means sending instructions to the IKBD and we discuss how to do this at the end.

8.2.3 The Mouse

Like the joystick, this has its own data packet handler. Again, we can write our own and substitute its address in the vector table. How this is done is shown in the file *systm_03.s*. This time the vector is number four in the table.

There is more than one way of setting up the mouse and we will use the most common which is called relative mode. There are three bytes in the mouse data packet and each one contains something useful. The first byte is *$f8* plus either (or both) of the two lowest bits set depending on which of the two buttons has been clicked. The second byte is the signed x-displacement and the third is the signed y-displacement since the last report. Signed means 2's complement so that for backward movements the top nibble is *$f*. But remember the displacement is measured in screen coordinates with the origin at the top left-hand corner of the screen and positive y is down.

8.3 Talking to the IKBD

Although the IKBD mainly concerns itself with reporting keyboard, joystick and mouse events, it is also able to receive instructions. These are necessary to configure it and select its reporting mode from the many options available. The way this is done is simple but not particularly obvious. It is done through the xbios call *IKBDWS* (intelligent keyboard write string) #25, in the following way. The instruction code to be sent to the IKBD is written as a string. The pointer to the string and the number of bytes minus one in the string are placed on the stack. Then the XBIOS function is called. How this is done for the specific cases of turning off the mouse, interrogating the joystick, turning off the joystick and setting up the mouse is shown in *systm_03.s*

8.4 Example Programs

8.4.1 *ramview.s*

This is the program to spy on the GEM variables area of RAM by making it the physical screen RAM. It uses the XBIOS routine SETSCREEN to do this. When the program is running you can operate the keyboard, joystick or mouse to see where pixels change, and calculate the approximate locations of the registers in RAM.

8.4.2 *key_peek.s*

Having found the approximate location of variables with ramview.s, now pin them down with the Devpac ST's monitor or debugger. Run this program (it's hardly a program at all) after having assembled it in the debugger. When the debugger is first entered switch to memory 3 (the block view) and modify the block address to something around *$e40*. Press CONTROL-Z to single step through the program and hold it down for a few seconds. This will fill the buffer and keep the program running for a short time. Whilst the program is running you can waggle the joystick, or whatever, and see which registers change. It will probably be necessary to search memory before and after this address to find the right area.

8.4.3 *systm_03.s*

This contains the subroutines for substituting ikbd data packet handlers and passing instructions to the IKBD.

Usually the joystick and mouse operate in the default mode of automatically reporting changes to their settings. Routines in this file stop this. When information is then required regarding these devices, it must be asked for. Part of the reason for doing it this way is to stop unwanted reporting from devices and the other is to illustrate how to communicate through the IKBD controller.

Note that communication with the controller is done in a roundabout way, with the address of the function number being pushed onto the stack before making the XBIOS call *IKBDWS* (write string).

8.4.4 *joy_test.s*

This uses the subroutines to show movements of the joystick. In particular it uses the VT52 terminal emulator routines in the operating system to write text. When the program is run, text will appear on the screen in response to movements of the joystick. The VT52 emulator routines provide a very powerful and simple method of displaying text in a variety of ways. A full list of functions available is given in Appendix 4.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                       *
*                            ramview.s                                  *
*                                                                       *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

* A look at what goes on in the GEM variables area
* Display the bottom 32K of user RAM on the screen
* While it's on view you can use the keyboard, joystick or mouse
* to see where data is being altered. The screen is 32k long and
* starts at 2049 so with a ruler you can measure where in RAM the
* variables are located.
* set up the screen

main    move.w  #-1,-(sp)       ignore resolution
        move.l  #2049,-(sp)     physical screen = bottom of user RAM
        move.l  #-1,-(sp)       forget the logical screen
        move.w  #5,-(sp)        xbios_setscreen
        trap    #14
        add.l   #12,sp          tidy
loop    bra     loop            idle
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                       *
*                          key_peek.s                                   *
* A program to find where the codes from the keyboard, joystick        *
* and mouse are kept.                                                   *
*                                                                       *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

* This is a very long program!
* Run it in the monitor or debugger and having selected the part of RAM
* to look at (around $e40) press the CONTROL and Z keys simultaneously.
* While the program is running you can also press any key or mouse button,
* or waggle the joystick and see where data appears. Then you know where
* to find it for your programs.

main    bra     main
        END
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                           *
*                          systm_03.s                                       *
* Joystick and mouse routines                                               *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

* End automatic reporting from the joystick
joy_off pea     joyoff_str              pointer the instruction string
        move.w  #0,-(sp)                1 less than length of string
        move.w  #25,-(sp)               function ikbdws
        trap    #14                     xbios
        addq.l  #8,sp                   tidy
        rts

* Interrogate the joystick
rd_joy  pea     rdjoy_str               ditto
        move.w  #0,-(sp)
        move.w  #25,-(sp)
        trap    #14
        addq.l  #8,sp
        rts

* Turn the mouse off
mse_off pea     mseoff_str              ditto
        move.w  #0;-(sp)
        move.w  #25,-(sp)
        trap    #14
        addq.l  #8,sp
        rts

* Set up the mouse for reporting in relative mode i.e. position change
mse_rel
        pea     relmse_str              ditto
        move.w  #0,-(sp)
        move.w  #25,-(sp)
        trap    #14
        addq.l  #8,sp
        rts

* Intercept GEM joystick routine with our own
init_joy:
        move.w  #34,-(sp)       find the table of vectors
        trap    #14             using xbios call kbvbase: address in d0
        addq.l  #2,sp           tidy stack
        move.l  d0,a0           base pointer
        move.l  24(a0),gem_joy  hijack GEM vector
        lea     joy_handle,a1
        move.l  a1,24(a0)       substitute mine
        rts                     and sneak off

* This is my joystick data packet handler. Now when an interrupt occurs
* my handler will be activated with a0 pointing to the data packet.
joy_handle:
        clr.w   d0
        move.b  1(a0),d0
        move.w  d0,joy_data     the second data byte has the info
        rts
```

```
* When I've finished, put back the GEM handler as if nothing happened
joy_term:
        move.w  #34,-(sp)       call xbios kbvbase
        trap    #14
        addq.l  #2,sp           tidy
        move.l  d0,a0           base pointer
        move.l  gem_joy,a1      dust off GEM's
        move.l  a1,24(a0)       and return it
        rts                     before I'm spotted


* Intercept the mouse packet handler with our own
init_mse:
        move.w  #34,-(sp)       find the table of vectors
        trap    #14             using xbios call kbvbase: address in d0
        addq.l  #2,sp           tidy stack
        move.l  d0,a0           base pointer
        move.l  16(a0),gem_mse  hijack GEM vector
        lea     mse_handle,a1
        move.l  a1,16(a0)       substitute mine
        rts                     and sneak off
* This is my mouse data packet handler. Now when an interrupt occurs
* my handler will be activated with a0 pointing to the data packet.
mse_handle:
        clr.w   d0
        move.b  (a0)+,d0        1st byte is the header
        move.w  d0,mse_click    save it
        move.b  (a0)+,d0        next byte is
        move.w  d0,mouse_dx     the x displacement relative to last position
        move.b  (a0),d0         last byte is
        move.w  d0,mouse_dy     ditto y
        rts

* When I've finished, put back the GEM handler as if nothing happened
mse_term:
        move.w  #34,-(sp)       call xbios kbvbase
        trap    #14
        addq.l  #2,sp           tidy
        move.l  d0,a0           base pointer
        move.l  gem_mse,a1      dust off GEM's
        move.l  a1,24(a0)       and return it
        rts                     before I'm spotted

* The strings to be sent to the ikbd are just the command numbers
* Each string is 1 byte long
mseoff_str      dc.b    $12     turn off the mouse
joyoff_str      dc.b    $15     turn off default automatic joystick reporting
rdjoy_str       dc.b    $16     interrogate the joystick
relmse_str      dc.b    $08     put the mouse in relative mode automatic report
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                        joy_test.s                                   *
* A routine to test the ikbd joystick function                        *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

* The joystick is interrogated and our own packet handler used to
* grab the data packet containing the FIRE bit (7) and the position
* bits (0-2) which is saved in the variable joy_data.

        opt     d+
        bra main
        include systm_03.s      the important subroutines
        even
main
* Set up joystick for standard reporting.
        bsr     init_joy        set up our packet handler
        bsr     joy_off         end automatic reporting
        bsr     mse_off         turn off the mouse
main1
        clr.w   joy_data
        bsr     rd_joy          read joystick
        move.w  joy_data,d0     here's the result
        move    d0,d1           save it
        andi.w  #$f0,d0         fire pressed  ?
        bne     fire_press      yes
        andi.w  #$f,d1          what direction is the stick?
        beq     joy_out         no direction
        cmp.w   #1,d1           up?
        beq     up              yes
        cmp.w   #2,d1           down?
        beq     down            yes
        cmp.w   #4,d1           left?
        beq     left            yes
        cmp.w   #8,d1           right?
        beq     right           yes
        bra     diagonal        only possibility left
* Use the VT52 subroutines for messages
up      pea     up_text         pointer to text
        move.w  #9,-(sp)        VT52 emulator
        trap    #1              GEM call
        addq.l  #6,sp           tidy
        bra     joy_out
down    pea     down_text
        move.w  #9,-(sp)
        trap    #1
        addq.l  #6,sp
        bra     joy_out
left    pea     left_text
        move.w  #9,-(sp)
        trap    #1
        addq.l  #6,sp
        bra     joy_out
right   pea     right_text
        move.w  #9,-(sp)
        trap    #1
        addq.l  #6,sp
        bra     joy_out
```

```
diagonal:
        pea     diag_text
        move.w  #9,-(sp)
        trap    #1
        addq.l  #6,sp
        bra     joy_out
fire_press:
        pea     fire
        move.w  #9,-(sp)
        trap    #1
        addq.l  #6,sp
joy_out clr.w   joy_data
        bra     main1

        SECTION BSS

gem_joy         ds.l    1
joy_data        ds.w    1
gem_mse         ds.l    1
mse_click       ds.w    1
mouse_dx        ds.w    1
mouse_dy        ds.w    1
        SECTION DATA
* Here are the messages to be printed. The number 27 is the ESCAPE
* code. In low res.Text can be positioned at any row (0 to 24)
* or column (0 to 39) but the number 32 must be added. Text must end in 0.
up_text dc.b    27,"E"          clear screen
        dc.b    27,"Y",33,50    type at row 1 (+32) and column 18 (+32)
        dc.b    "up"            the word "up"
        dc.b    0               end of text
down_text:
        dc.b    27,"E"
        dc.b    27,"Y",56,50
        dc.b    "down"
        dc.b    0
left_text:
        dc.b    27,"E"
        dc.b    27,"Y",44,32
        dc.b    "left"
        dc.b    0
right_text:
        dc.b    27,"E"
        dc.b    27,"Y",44,62
        dc.b    "right"
        dc.b    0
fire    dc.b    27,"E"
        dc.b    27,"Y",44,50
        dc.b    "FIRE"
        dc.b    0
diag_text:
        dc.b    27,"E"
        dc.b    27,"Y",44,50
        dc.b    "diagonal"
        dc.b    0
        END
```

9

Hidden Surfaces and Illumination

A computer is a fast number cruncher, but it doesn't know anything about the real world. When it comes to conveying simple everyday experiences like not being able to see through solid opaque objects, the computer is a real loser. There are no codes in the processor instruction set which allow us to easily convey such information. It seems obvious to us that the rear sides of opaque objects are not visible and that an opaque object will obscure those behind it. Making the computer show this simple fact of life is hard work. It is called the hidden surface problem and it is the basis of some very time-consuming algorithms in computer graphics.

For any micro without dedicated graphics hardware, this becomes a severe problem since the burden of computation falls on the main processor, and of necessity therefore, any strategy we adopt to deal with hidden surfaces cannot be too time consuming. As a consequence, the geometry of the objects themselves cannot be so complex as to require a time consuming hidden surface algorithm. The simplest solution is to require that all polyhedra be convex, i.e. each surface polygon looks outward and not towards another polygon. It is possible to deal with simple polyhedra which are not convex but we shall only consider ones which are convex. It is always possible to construct complex objects out of several convex polyhedra and the strategy then is to draw the furthest ones first and the nearest ones last. This is the so called 'painter' algorithm by which objects in the background are naturally obscured by those in the foreground. More of this later.

The procedure for deciding whether a surface is visible, combines naturally with the calculation to decide how brightly it is illuminated by a distant light source, a necessary attribute if the object is to look real. Surfaces which face towards the

light source must be brighter than those which face away. We shall combine both of these into a single algorithm in this chapter.

9.1 Hidden Surface Removal

In the simple strategy for convex polyhedra adopted here, deciding whether a surface is visible requires a substantial amount of vector algebra (which can be minimised by pre-calculating certain surface parameters) . The procedure is straightforward: a polygonal surface is visible if it faces the view point. The problem is how to convert the word "faces" into a mathematical expression. This is done in the following way.

Each surface has associated with it a vector which points out at right angles from the surface so that the polyhedron as a whole looks like a porcupine. All such vectors have the same length, which is chosen to be unity. They are called surface normal unit vectors. The only difference between two unit vectors is their direction, which reflects the different directions in which the surfaces face as shown in Figure 9.1. Of course, for the purposes of calculation, 1 is not a useful size for a vector and so it is multiplied by the factor 16384 (2^{14}). This keeps quantities within word size and makes multiplication and division simple.

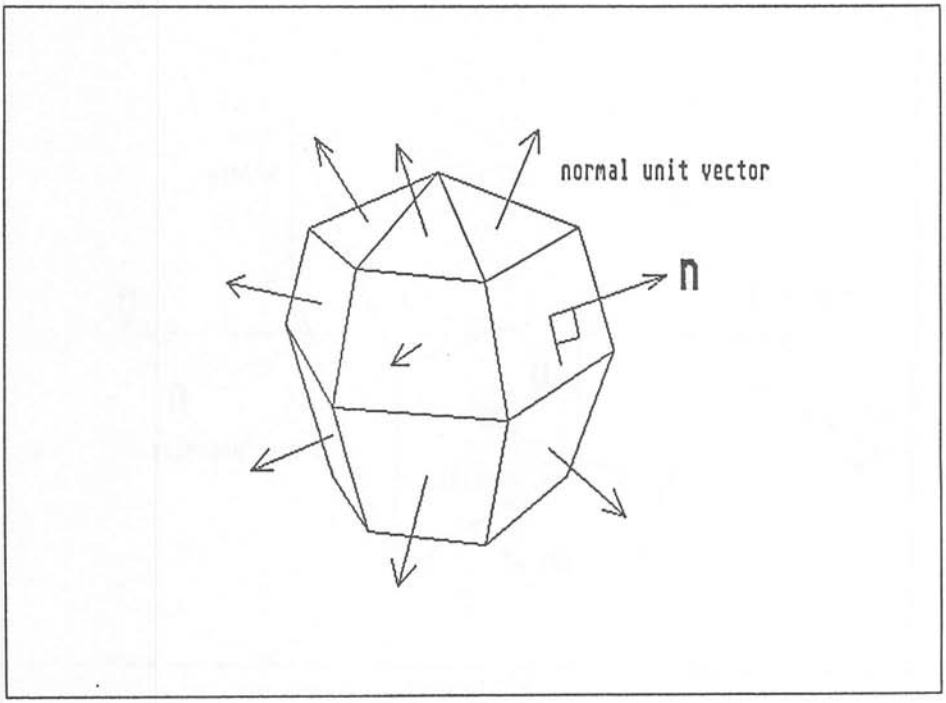

Figure 9.1 A convex polyhedron showing surface normal vectors

To see whether a surface is visible from the view point now consists of testing whether its unit vector is in the same or opposite direction to a vector (the view vector) drawn from the viewpoint to the surface. There is a basic vector product which performs this test. It is called the scalar or dot product. Appendix 6 explains products involving vectors. In the language of mathematics, where the view vector is **V** and the surface normal vector is **n,** the scalar product will yield a positive result if the surface is hidden and a negative result if it is visible:

hidden: scalar product **V.n** is positive

visible: scalar product **V.n** is negative.

The scalar product itself is really nothing more than the distance from the view point to the surface times the cosine of the angle between the view vector and the surface normal. The sign of the product naturally follows therefore from the fact that the cosine of an angle less than 90° is positive whereas the cosine of an angle between 90° and 180° is negative. Figure 9.2 shows the directions of the vectors for a visible and a hidden surface. All this is very satisfactory except for one thing; the surface normal unit vector must be calculated and that is not so simple. Here the unit vector is calculated in view frame coordinates.

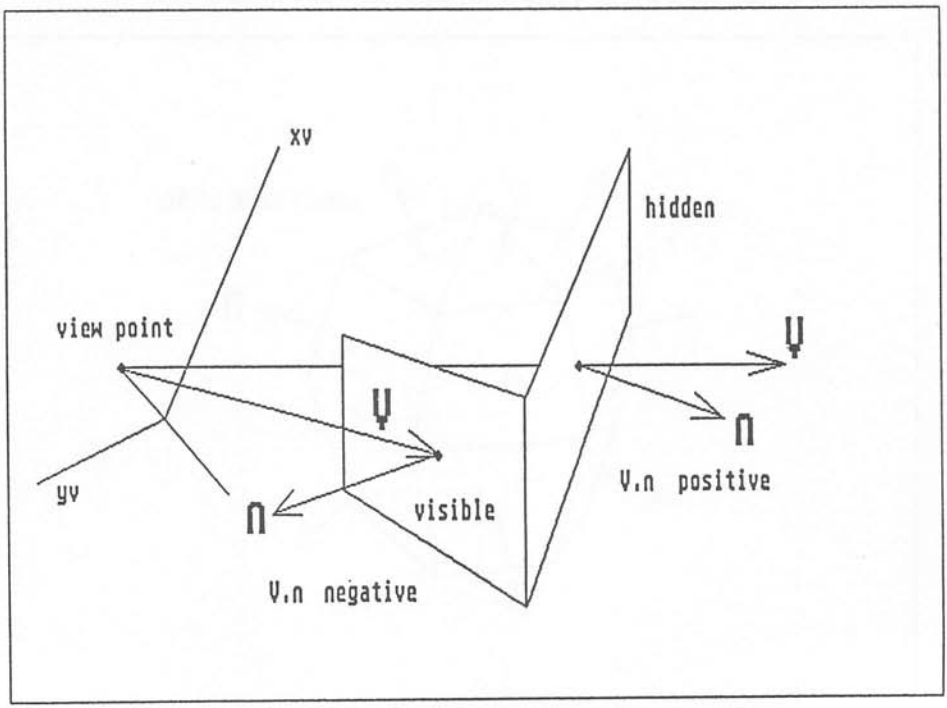

Figure 9.2 Visible and hidden surfaces

As a brief digression, it's worth mentioning that the test for visibility can be done without any reference to vector products. The way that data lists have been set up, with the list of edge connections of a polygon going clockwise when viewed from the front, can be used to give a simple test for visibility. When converted to screen coordinates by the perspective transform, visible polygons have their edge list going anticlockwise. Projected polygons with clockwise screen edge lists will therefore have come from polygons facing away from the screen and which should be hidden. A test for this can easily be constructed.

We choose to use the scalar product here because the normal unit vectors, once calculated, can also be used to determine the level of illumination of each surface.

9.2 Calculating the Surface Normal Unit Vector

The procedure to calculate the normal unit vectors requires quite a lot of vector algebra and time consuming multiplications. It can be minimised by working out some relevant quantities beforehand and storing the data in a list in the usual way. In fact the normal vectors themselves could be completely worked out in the object frame and transformed together with the vertices at each stage. There are substantial advantages to doing it this way.

Instead, we choose to calculate the vectors in view frame coordinates because of the way it fits in nicely with the evolution of our program and the tutorial objective of the book. The particular vector product which allows us to calculate the normal vector is called a cross product. It's more difficult to understand than the scalar product but it's precisely what we want. Appendix 6 also covers this topic.

A vector product is illustrated in Figure 9.3. for a single polygon. Going round the perimeter of the polygon, the first two edges we meet are from vertices 1 to 2 and 2 to 3. Let us call the vectors associated with these edges **A**12 and **A**23. The normal vector **B** is then calculated as the cross product between them:

B = **A**23 x **A**12.

This shorthand notation is all fairly meaningless until translated into a set of mathematical operations. The x, y and z components of A12 and A23 are:

A12x = x2-x1, A12y = y2-y1, A12z = z2-z1

A23x = x3-x2, A23y = y3-y2, A23z = z3-z2

and the components of **B** are:

Bx = A12z.A23y-A12y.A23z

By = A12x.A23z-A12z.A23x

Bz = A12y.A23z-A12x.A23y

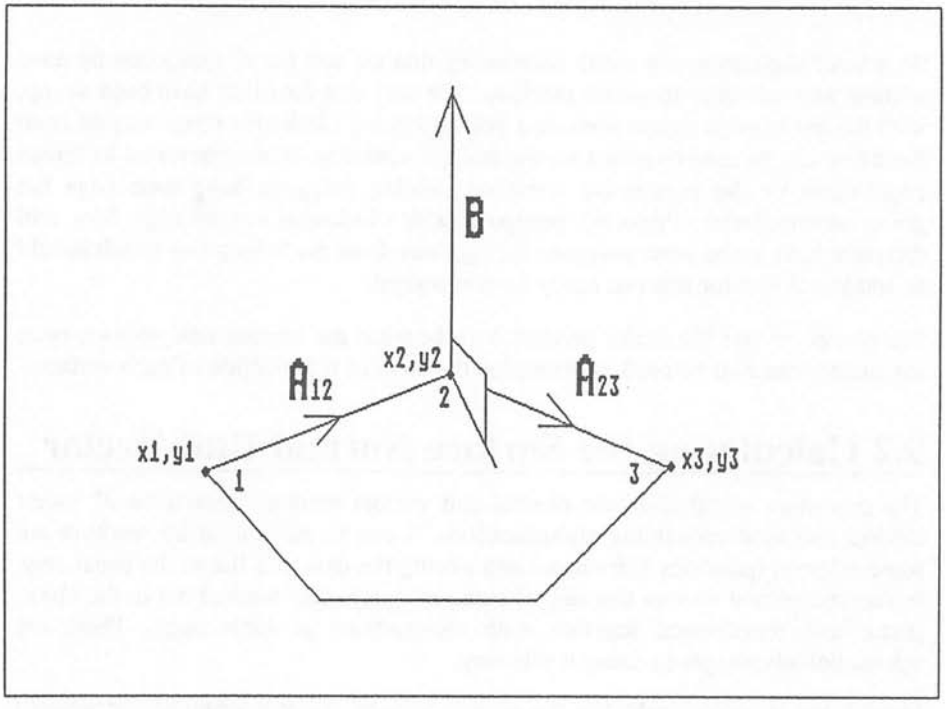

Figure 9.3 The vector product of two vectors

These multiplications constitute the bulk of the calculation.

There is one final step. What we want is the unit vector. The vector B is in the right direction but its size is too large. To get the unit vector, each of the components must be divided by the magnitude of **B**. This provides an additional chore because the magnitude of **B** is calculated from:

$$B = \sqrt{(Bx^2+By^2+Bz^2)}$$

which requires taking a square root. How this is done is explained in the example program.

Once the magnitude B has been calculated, the components of the unit vector are

$$bx = Bx/B,\ by = By/B,\ bz = Bz/B.$$

After this the line-of-sight vector (view vector) from the view point to the first vertex of the surface in the edge list is then found and the scalar product taken with the normal vector. On the basis of this test, the surface is either flagged as hidden or else its level of illumination calculated. We discuss illumination next.

9.3 Illumination and Colour

It is possible to employ the most elaborate computations to construct geometrically accurate 3D models, and yet the attributes which make them look real may be very subtle and less obvious. In sprite graphics, the shadow on the ground which follows the motion of a projectile is a small but essential clue to its altitude. In 3-D, one of the easiest and dramatic improvements to add realism to a model is illumination by a light source. Facets which face the light source are more brightly illuminated than those which face away. As the object changes its orientation, so the changes in illumination give additional visual clues to its shape and structure. This is what we shall try to simulate next. There are limitations to what can be achieved on a the ST, not so much a consequence of software constraints, but mainly resulting from the way colour is implemented in the colour palette. The way in which illumination is determined is very similar to the way visibility is tested for, but in this case an actual number must be generated, depending on the angle of the surface to the light source.

The direction of the beam of light emanating from a light source is specified by a vector, called the illumination vector. It would be possible to simulate a diverging or converging beam by having this vector change its direction across the field of illumination, but for simplicity the beam is taken to be parallel. Consequently a single vector is sufficient to define to direction of the beam. Likewise, the intensity of the light is taken to be constant everywhere. These approximations are valid for a distant light source such as the Sun, but the difference for a near light source is hardly noticeable. This illumination vector is also a unit vector, (i.e. it has a magnitude of unity.)

Because we have already calculated the surface normal unit vectors, everything is set up to find the level of illumination of each facet on the surface. Figure 9.4 illustrates the calculation. It is nothing more than the scalar product of the illumination vector and the normal vectors. This is a realistic calculation since the level of illumination does depend on the cosine of the angle between the two vectors.

There is one minor modification we will use in the calculation. Consider how the earth is illuminated by the Sun: the side which faces the Sun is brightly lit but the side which faces away would be pitch black if it weren't for the reflected light of the Moon (forgetting the light from the stars). In a room a single light source is sufficient to illuminate everything, though much of this is back-reflected light from the walls and all the objects in the room. This is the basis of the Radiosity method of illumination calculation which is used in very advanced graphics to simulate realism to a high degree. We can incorporate a very rudimentary version of this into our method, using the scalar product to set an illumination level even where it is negative, so there is some illumination even on the dark side of objects.

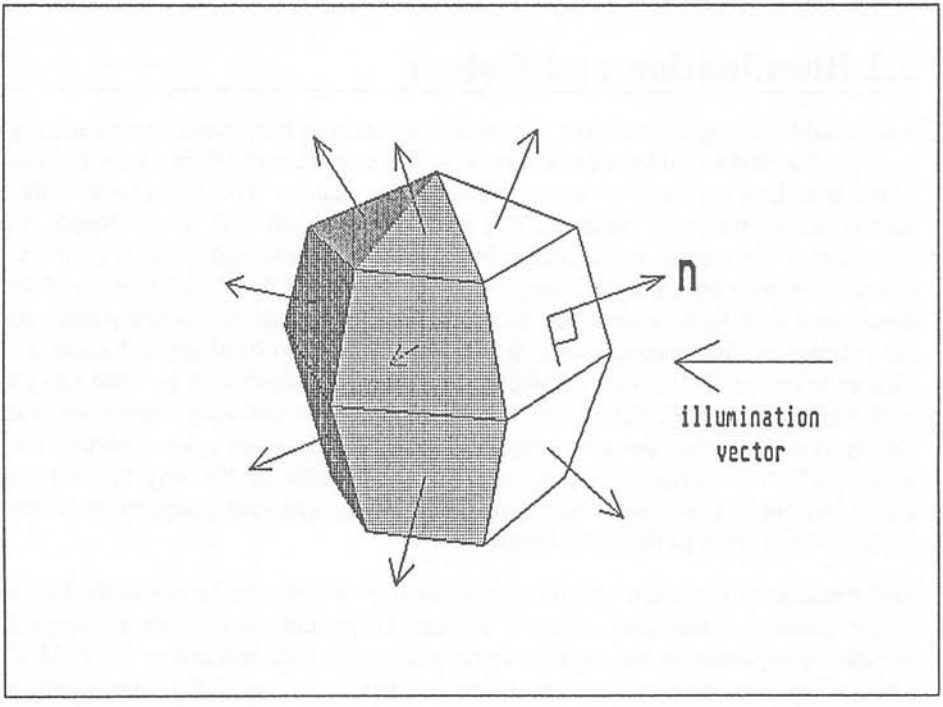

Figure 9.4 Surface illumination

Here then is the method in outline: for each surface, take the scalar product of the illumination vector with the normal unit vector; since both vectors are 1 in magnitude, this will yield a result between +1 (minimum illumination) and -1 (maximum illumination). If you're confused by the sign, remember in our geometry the illumination vector points away from the light source. Since in our method all unit vectors are multiplied by 2^{14}(16384), the scalar product will actually yield a result somewhere in the range -2^{28} to $+2^{28}$. Adding 2^{28} to this result and dividing by 2^{25}(by right shifting) reduces this to the range 0 to 16. This result can then be used to index 16 different colour shades. How this is done requires a brief explanation of the colour palette.

9.3.1 The Colour Palette

In low resolution, which is the most colourful, 16 different colours can be displayed simultaneously out of a possible 512. This selection of 16 is called the colour palette. There are tricks to exceed 16 for the screen as a whole by changing the colour palette frequently whilst a picture is being drawn (during the horizontal blank, for example). We will use the basic 16. For what follows Figure 9.5 will be of assistance. The ST standard palette settings are listed in Appendix 8.

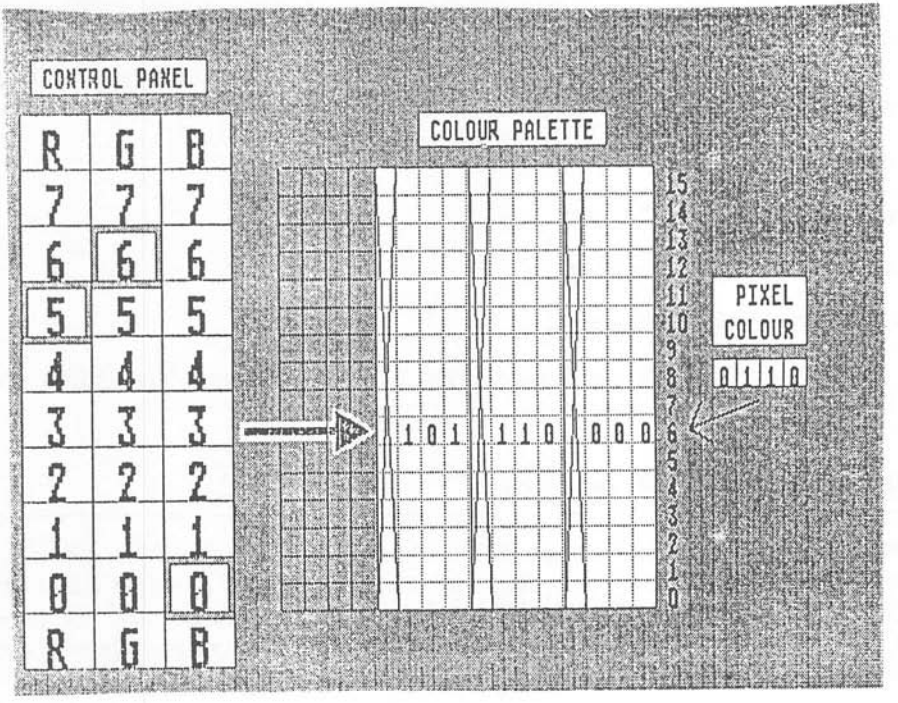

Figure 9.5 The control panel and the colour palette

An excellent aid to understanding how the colour palette works is found in the Control Panel Accessory which comes with the ST. This shows three sliders of red, green and blue, each with 8 possible settings in low resolution. This means there are 8.8.8 = 512 possible combinations. Have a play with the sliders to see what colours can be obtained. At any one time 16 of these 512 can be displayed on the screen simultaneously. Why sixteen? Because there are 4 colour planes in low resolution, as we have seen in Chapter 2, and each plane is represented by a bit in the colour nibble so that up to 16 combinations are available. The value of the colour nibble is used to index a 'pot' from the colour palette which contains the word number of the colour. Simple isn't it!

All that remains is to find out how to generate the colour word in the palette from the red, green and blue settings in the control panel. In fact the nibbles follow exactly as they are presented in the control panel when written in hexadecimal. A setting of *$0777* (white) means red=7, green=7 and blue=7. If you want to write them in decimal, the recipe is:

colour value = 256*(red setting) + 16*(green setting) + 1*(blue setting)

The chosen colours must then be loaded into the palette with an Operating System call. This is what is done in the example program.

For our purposes, in order to simulate lighting, the colours will be different shades of the same colour. There is obviously going to be a trade off here. With a maximum of 16 colours the following combinations are possible:

mode 0 16 shades of one colour

mode 1 8 shades of 2 colours

mode 2 4 shades of 4 colours

mode 3 2 shades of 8 colours.

The last of these isn't worth considering but the other three possibilities are implemented in the programs.

9.4 Example Programs

The example programs show the ST monolith in rotation with hidden surface removal and illumination. The program is set up with rotation about the x axis but this can be altered as desired. The monolith is coloured in red and blue but, once again, it is good fun to set up alternative palettes in different colours following the colour recipe, above.

9.4.1 *ill_hide.s*

This is the control program. It still uses the data for the ST monolith to display it rotating about any, or all three of the object frame axes. Because we now have hidden surface removal, it doesn't matter if the angles become large enough to display the back. Nothing will be displayed because the back is hidden. The program is set for rotation about the x-axis of the object frame.

The colour palette has been set up to use 7 shades of blue and 8 shades of red. The first colour in the palette has the value 0 which is black and is used by the system to provide the background. The shading mode is flexible and is set up by means of a key, called *illkey*, which has a value equal to the mode number, above. The program is set up in mode 1.

9.4.1 *core_04.s*

This calculates surface normal vectors, determines whether a surface is visible and if so calculates the level of illumination and the final palette colour as outlined in the text. Because of the limitations of word multiplication in the calculation of normal vectors, objects are restricted to linear dimensions of less than about 200.

First of all the surface normal vectors are calculated as described above. In the subroutine *nrm_vec* the normal vector is converted to a unit vector by dividing

each of its components by the magnitude of the vector. The magnitude is calculated by Pythagoras' theorem in 3D and requires a square root operation which is done in the subroutine *sqrt* by an iterative process.

The square root algorithm works in the following way. Suppose the square root of a number, N, is known approximately; call it sqrt1. Then a better approximation, sqrt2, can be found by dividing the number by sqrt1, adding this to sqrt1 and dividing by 2, i.e.

$$sqrt2 = 1/2(sqrt1 + N/sqrt1).$$

Sqrt2 is a better approximation than sqrt1. Then starting with sqrt2 an even better approximation, sqrt3, can be found in the same way. Each one of these recalculations is called an iteration. Starting with a modest approximation, only three iterations are needed in the routine to calculate a square root accurate to 1 part in 2^{16}, i.e. as accurate as a word will allow.

The line-of-sight vector used to determine visibility in *vis_ill* is taken from the view point to the first vertex on a surface. There is no ambiguity here since at the point where a surface just ceases to be visible all vertices give a line-of-sight vector perpendicular to the surface normal. The illumination vector is specified by its components *ill_vecx*, *ill_vecy* and *ill_vecz* each multiplied by 2^{14} for accuracy, as usual.

If a surface is invisible, the illumination is set to the value *$f0*. Otherwise the intrinsic colour, 0 or 1 in mode 1 (the mode used here), is then combined with the shading to produce a number to index the colour palette. This is a tricky calculation and best understood by following the algorithm through.

Depending on whether the colour is 0 or 1, either the colours fron 1 to 7 (blue) (0 is reserved for black, the background) or from 8 to 15 (red) are selected. The actual shading level then fixes which colour in the group is chosen, with the lightest being 1 (blue) and 8 (red) and the darkest being 7 (blue) and 15 (red).

The colour palette is set up by XBIOS call number 6, with a pointer to the list of colours.

9.4.3 *data_04.s*

This contains the illumination vector components, which in this example define a light source shining from right to left in the view frame. This is clearly no good in general since the light source should be fixed in the world frame and transformed like everything else to the view frame.

Following this are the intrinsic colours (red or blue in this case) corresponding to the two posibilities, 0 or 1, in mode 1. The colours for the palette are listed in

hexadecimal, as the settings appear on the control panel, giving 7 shades of blue and 8 shades of red.

9.4.4 *bss_04.s*

Additional variables from Chapter 9.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                          ill_hide.s                                   *
* A program to illustrate illumination and hidden surface removal      *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

        SECTION TEXT
        opt     d+
        bra main
        include systm_02.s      housekeeping file
        include core_03.s       subroutines
        include core_04.s       illumination, hidden surface removal

main    bsr     find_screens    find the addresses of the two screens
        bsr     wrt_scrn1_tbl   write a row address table for screen1
        bsr     wrt_scrn2_tbl   ditto                          screen2
        bsr     hline_lu
        bsr     hide_mse        exterminate the mouse
        bsr     palette_set     set up the shades of blue and red

* transfer all the data from my lists to program lists
        bsr     transfer
* place it in the world frame
        move.w  #0,Oox          on the ground
        move.w  #100,Ooz        100 in front
        clr.w   Ooy             dead centre
* Initialize angles for rotation
        clr.w   otheta
        move.w  #50,ophi        tilt it forward
        clr.w   ogamma
* Initialize screens
        clr.w   screenflag      0=screen 1 draw, 1=screen 2 draw
        bsr     clear1          clear the screens
        bsr     clear2

* Start the rotation about the xw axis
loop5   move.w  #360,d7         a cycle
loop4   move.w  d7,otheta       next theta
        move.w  d7,-(sp)        save the angle
        tst.w   screenflag      screen 1 or screen2?
        beq     screen_1        draw on screen 1, display screen2
        bsr     draw2_disp1     draw on screen 2, display screen1
        bsr     clear2          but first wipe it clean
        clr.w   screenflag      and set the flag for next time
        bra     screen_2
screen_1:
        bsr     draw1_disp2     draw on 1, display 2
        bsr     clear1          but first wipe it clean
        move.w  #1,screenflag   and set the flag for next time
screen_2:
        bsr     otranw          object-to-world transform
```

```
* pass on the new coords
        move.w  oncoords,d7
        move.w  d7,vncoords
        subq.w  #1,d7
        lea     wcoordsx,a0
        lea     wcoordsy,a1
        lea     wcoordsz,a2
        lea     vcoordsx,a3
        lea     vcoordsy,a4
        lea     vcoordsz,a5
loop6   move.w  (a0)+,(a3)+
        move.w  (a1)+,(a4)+
        move.w  (a2)+,(a5)+
        dbra    d7,loop6

* Test for visibility and lighting
        bsr     illuminate      if it's visible find the shade
* Complete the drawing
        bsr     perspective     perspective
        bsr     polydraw        finish the picture
        move.w  (sp)+,d7
* Check for termination
        bsr     scan_keys       has a key been pressed?
        tst     d0              if so
        bne     term            back to caller
        sub.w   #10,d7          otherwise increment in 10 degree steps
        bgt     loop4           next angle
        bra     loop5           or repeat
term    clr.w   -(sp)           terminate and
        trap    #1              back to caller

        SECTION DATA
        include data_01.s
        include data_03.s
        include data_04.s
        SECTION BSS
        include bss_03.s
        include bss_04.s
        END
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                        core_04.s                                      *
*                   Subroutines for chapter 9                           *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

illuminate:
* New subroutines:
* calc_nrm - calculate the polygon normal unit vectors
* calc_ill - calculate the level of illumination 0 - 7
* vis_ill  - convert this to a palette colour
* transfer - move my data to program data
* Calculate the normal unit vectors. All components are *2^14 for accuracy
calc_nrm:
        move.w  npoly,d7        any to do?
        beq     nrm_out         quit if none
        subq    #1,d7           ready to loop
        lea     vcoordsx,a0     coords
        lea     vcoordsy,a1
        lea     vcoordsz,a2
        lea     sedglst,a3      connections
        lea     snedges,a4      no. edges per poly
        lea     snormlst,a5     surface normals pointer
* Calculate the surface normal unit vectors
next_nrm:
        move.l  a5,-(sp)        save pointer to normals list
        move.w  (a3),a5 first vertex of the next surface
        move.w  2(a3),a6        second vertex
        add     a5,a5           *2 for offset
        add     a6,a6           ditto
        move.w  0(a0,a6.w),d1   x2
        sub.w   0(a0,a5.w),d1   x2-x1 = A12x
        move.w  0(a1,a6.w),d2   y2
        sub.w   0(a1,a5.w),d2   y2-y1 = A12y
        move.w  0(a2,a6.w),d3   z2
        sub.w   0(a2,a5.w),d3   z2-z1 = A12z
        move    a6,a5
        move.w  4(a3),a6        third vertex
        add     a6,a6           *2 for offset
        move.w  0(a0,a6.w),d4   x3
        sub.w   0(a0,a5.w),d4   x3-x2 = A23x
        move.w  0(a1,a6.w),d5   y3
        sub.w   0(a1,a5.w),d5   y3-y2 = A23y
        move.w  0(a2,a6.w),d6   z3
        sub.w   0(a2,a5.w),d6   z3-z2 = A23z

        movea.w d2,a5           save
        muls    d6,d2
        movea.w d3,a6           save
        muls    d5,d3           ditto
        sub.l   d2,d3           Bx
        move.l  d3,-(sp)        save it on stack
        move.w  a5,d2           restore
        move.w  a6,d3           restore
        movea.w d3,a5           save
        muls    d4,d3
        movea.w d1,a6           save
        muls    d6,d1
        sub.l   d3,d1           By
        move.l  d1,-(sp)        save it
        move.w  a6,d1           restore
```

```
* last component, no need to save values
        muls    d5,d1
        muls    d4,d2
        sub.l   d1,d2           Bz
        move.l  d2,-(sp)        save it

        movem.l (sp)+,d4-d6     Bx in d6, By in d5 and Bz in d4
nrm_cmpt:
        lsr.l   #2,d4           /4 to prevent overspill
        lsr.l   #2,d5
        lsr.l   #2,d6
        move.w  d4,d0
        move.w  d5,d1
        move.w  d6,d2
        move.l  d7,-(sp)        save
        bsr     nrm_vec         calculate the unit vectors bx, by, bz
        move.l  (sp)+,d7        restore
        move.w  d0,d4
        move.w  d1,d5
        move.w  d2,d6
        move.l  (sp)+,a5        restore pointer to normals list
        move.w  d6,(a5)+        save nx
        move.w  d5,(a5)+        save ny
        move.w  d4,(a5)+        save nz

        move.w  (a4)+,d0        number of vertices in this surface
        addq    #1,d0           the edge list always repeats the first
        add     d0,d0           *2 for offset
        adda.w  d0,a3           adjust edge list pointer to next surface
        dbra    d7,next_nrm     do all the surfaces (polygons)
nrm_out:

vis_ill:
* Find the visibility and level of illumination of a surface by taking
* the scalar product of the surface unit normal vector with the
* line of sight vector from the viewpoint and with illumination vector
* respectively.
        move.w  npoly,d7
        subq    #1,d7
        lea     vcoordsx,a0
        lea     vcoordsy,a1
        lea     vcoordsz,a2
        lea     snedges,a3
        lea     sedglst,a4
        lea     snormlst,a5     surface unit normals list
        lea     slumlst,a6      surface illumination and visibility list
        move.w  ill_vecx,d0     illumination vector x-component
        move.w  ill_vecy,d1     ditto   y
        move.w  ill_vecz,d2     ditto   z
```

```
* The line-of-sight vector is taken between the first vertex on the
* surface and the view point.
next_ill:
        move.w  (a4),d6         1st point on next surface
        add     d6,d6           offset
        move.w  0(a0,d6.w),d3   is the line-of sight x-cmpt., xls
        move.w  0(a1,d6.w),d4   yls
        move.w  0(a2,d6.w),d5   z
        sub.w   vwpointz,d5     zls :view point lies along -zv axis
        muls    (a5),d3         nx*sx
        muls    2(a5),d4        ny*sy
        muls    4(a5),d5        nz*sz
        add.l   d4,d3
        add.l   d5,d3           scalar product
        bmi     visible         is negative if surface visible
* It is hidden
        move.w  #$f0,(a6)+      set illumination for hidden and move on
ill_tidy:
        addq.w  #6,a5           update normals pointer
        move.w  (a3)+,d5        current no. edges
        addq    #1,d5           first vertex is repeated
        add     d5,d5           2 bytes/word
        adda.w  d5,a4           update edge list pointer
        dbra    d7,next_ill             for all surfaces
        bra     set_colr                go on to set the colours

* The surface is visible so find the illumination level.
* Remember all vectors are *2^14
visible:
        move.w  d0,d3           copy the illumination vector
        move.w  d1,d4
        move.w  d2,d5
        muls    (a5),d3         nx*illx
        muls    2(a5),d4        ny*illy
        muls    4(a5),d5        nz*illz
        add.l   d4,d3
        add.l   d5,d3           -2^28 < scalar product < +2^28
        add.l   #$11100000,d3   0 < scalar product < 2^29
        move.w  #25,d4
        lsr.l   d4,d3
        cmp.w   #$f,d3          keep in range 0 to $f
        ble     vis_1           correct
        move.w  #$f,d3          for
        bra     ill_save        errors
vis_1   cmp.w   #0,d3
        bge     ill_save
        clr     d3
ill_save:
        move.w  d3,(a6)+        save it
        bra     ill_tidy        next one
```

```
set_colr:
* The illumination level is combined with the intrinsic colour to produce
* the final displayed colour.
* Illkey is used to determine the number of shades per colour so that
* different lighting levels can be simulated:
* illkey = 2 gives 4 shades of 4 colours: 0, 1, 2, 3
* illkey = 1 gives 8 shades of 2 colours: 0, 1
* illkey = 0 gives 16 shades of 1 colour: 0
        move.w  npoly,d7
        subq.w  #1,d7           the counter
        move.w  illkey,d0       how many shades per colour
        lea     slumlst,a0      the levels of illumination
        lea     srf_col,a1      raw intrinsic colours: 0 or 0,1 or 0,1,2,3
        lea     col_lst,a2      final colours for display
        moveq.w #4,d6
        sub.w   d0,d6           4-illkey
next_col:
        move.w  (a0)+,d1        next illumination
        cmp.w   #$f,d1          is it hidden?
        ble     set_col         no
        move.w  #$f0,(a2)+      it is, set flag
        addq.l  #2,a1           point to next intrinsic colour
        bra     set_next        and go on
set_col lsr.w   d0,d1           divide by 0, 2, or 4
        move.w  (a1)+,d2        the intrinsic colour
        rol.b   d6,d2           0 or 0,8 or 0,4,8,12 = colour base
        add.w   d1,d2           illumination + colour base
        bgt     pass_col
        moveq.w #1,d2           avoid background
pass_col:
        move.w  d2,(a2)+        = final colour
set_next:
        dbra    d7,next_col     for all surfaces
        rts
* Set the colour palette
palette_set:
        pea     palette         here's my palette
        move.w  #6,-(sp)        setpalette function
        trap    #14             xbios
        add.w   #6,sp           tidy
        rts
* Transfer my data to program data
transfer:
        move.w  my_npoly,d7     no. of polygons
        move.w  d7,npoly        pass it
        subq.w  #1,d7           the counter
        move.w  d7,d0           save it
        lea     my_nedges,a0    source
        lea     snedges,a1      destination
        lea     intr_col,a2     my intrinsic colours
        lea     srf_col,a3      program intrinsic colours
loop0   move.w  (a0)+,(a1)+     transfer edge nos.
        move.w  (a2)+,(a3)+     transfer intrinsic colours
        dbra    d0,loop0
```

```
* Calculate the number of vertices altogether
        move.w  d7,d0           restore count
        lea     my_nedges,a6
        clr     d1
        clr     d2
loop1   add.w   (a6),d1         no more than this
        add.w   (a6)+,d2        total no. of vertices
        addq    #1,d2           and with last one repeated each time
        dbra    d0,loop1
* Move the edge list
        subq    #1,d2           the counter
        lea     my_edglst,a0    source
        lea     sedglst,a1      destination
loop2   move.w  (a0)+,(a1)+     pass it
        dbra    d2,loop2
* and the coords list
        move.w  d1,oncoords
        subq    #1,d1           the counter
        lea     ocoordsx,a1
        lea     my_datax,a0
        lea     ocoordsy,a3
        lea     my_datay,a2
        lea     ocoordsz,a5
        lea     my_dataz,a4
loop3   move.w  (a0)+,(a1)+
        move.w  (a2)+,(a3)+
        move.w  (a4)+,(a5)+
        dbra    d1,loop3
* and the window limits
        move.w  my_xmin,xmin    ready
        move.w  my_xmax,xmax    for
        move.w  my_ymin,ymin    clipping
        move.w  my_ymax,ymax
        rts
nrm_vec
* normalise a vector: unormalised components in d0,d1,d2
* normalised components returned
        move    d0,d3           save
        move    d1,d4           the
        move    d2,d5           components
        muls    d0,d0           squares
        muls    d1,d1
        muls    d2,d2
        add.l   d1,d0
        add.l   d2,d0           sum of squares
        bsr     sqrt            calculate the magnitude
        move.w  #14,d7          multiply
        ext.l   d3              the
        ext.l   d4              components
        ext.l   d5              by
        lsl.l   d7,d3           2^14
        lsl.l   d7,d4
        lsl.l   d7,d5
        divs    d0,d3           divide by
        divs    d0,d4           the magnitude
        divs    d0,d5           to derive
        move.w  d3,d0           the
        move.w  d4,d1           normalised
        move.w  d5,d2           components
        rts
```

```
sqrt:
* A routine to find the square root of a long word N in d0
* in three iterations using the formula
* squrt = 1/2(squrt + n/squrt)
* An approximate starting value is found from the highest bit in d0
* Result passed in d0.w
        tst.l   d0
        beq     sqrt2           quit if 0
        move.w  #31,d7          31 bits to examine
sqrt1   btst    d7,d0           is this bit set?
        dbne    d7,sqrt1
        lsr.w   #1,d7           this bit is set and 2^d7/2 is approx root
        bset    d7,d7           raise 2 to this power
        move.l  d0,d1           N
        divs    d7,d1           n/squrt
        add     d1,d7           squrt+N/squrt
        lsr.w   #1,d7           /2 gives new trial value
        move.l  d0,d1           N
        divs    d7,d1
        add     d1,d7
        lsr.w   #1,d7           second result
        move.l  d0,d1
        divs    d7,d1
        add     d1,d7
        lsr.w   #1,d7           final result
        move.w  d7,d0
sqrt2   rts
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                       data_04.s                             *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

ill_vecx        dc.w    0
ill_vecy        dc.w    -16384  light shining from +y to -y
ill_vecz        dc.w    0
vwpointz        dc.w    -100
illkey          dc.w    1
intr_col        dc.w    0,1,0,0,1,1
palette dc.w    0,$557,$446,$336,$226,$225,$114,$113
        dc.w    $756,$745,$734,$723,$713,$702,$502,$401
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                       bss_04.s                                        *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* Variables for surface illumination and colour
snormlst        ds.w    100
slumlst         ds.w    40
srf_col         ds.w    40
```

10

General Transforms in 3D

In this chapter we investigate a number of transforms of various kinds involved in the manipulation of 3D structures.

10.1 Geometric Transforms

Combinations of simple rotations and displacements are extensively used in the construction of a complex scene consisting of several graphics primitives in different locations and with different orientations. Besides these instance transforms, there are other more exotic distortions that can be used. Structures can be manipulated in a variety of ways:

rotation - a change of orientation,

shear - distortion,

scaling - change in size,

reflection - replacement by a mirror image,

inversion - inside out and back to front,

In general, any 3x3 matrix will produce a combination of scaling and shear. In the special case that there is no change in volume, what results is a pure rotation. Sometimes shears with fixed (simple) matrix elements are used to simulate rotation by fixed angles. The first three of these transforms are illustrated in this chapter, with input and control from the keyboard and joystick.

Transformations of these kinds are easily implemented using matrices and several of them can be combined by concatenation (multiplication) of the individual matrices prior to actually transforming the points. Where a large number of points

is concerned, this saves a lot of time compared to performing each transform separately. An example of this is shown in the programs.

10.1.1 Rotations

When the joystick is moved or a key is pressed we want to see a corresponding rotation on the screen. In principle, doing this is very simple. For example, a movement of the joystick to the left could cause a positive rotation about the x-axis and a movement to the right could cause a negative rotation. Other joystick movements could produce rotations about other axes. The matrices for simple rotations about the x, y, and z axes have all been listed in Chapter 7.

Following each movement of the joystick, a new set of object vertices could be generated by multiplying the old vertices by the appropriate rotation matrix. In this way the results of the previous rotation would be used as the starting point for the next. The problem with doing this is that errors in the accuracy with which binary arithmetic is done in the transformations accumulate from frame to frame and eventually reduce the picture to chaos. A solution to this problem is to redraw the object each time from a reference position (like the object frame) with information stored in a set of "signposts" (unit vectors again) which have been continuously rotated with the object to keep up with joystick movements. Then the object is only transformed once each time. This method is essential in the viewing transform when the observer is moving freely. This is discussed extensively in the next chapter.

Alternatively, there is a simple way to implement rotations, but with a motion determined by a scheme similar to that involving lines of longitude and latitude, where rotations about the y and x axes are added up separately and finally put together at the end. In this scheme, several movements of the joystick (say) may have taken place both left or right (rotation about the x axis) and up and down (rotation about the y axis) in any order, but only the separate totals are recorded. A single movement of the joystick may correspond to a 1° increment in that direction.

As an example, suppose the total rotation about the y-axis is 40° and the total rotation about the x-axis is 83°. Then the overall rotation is taken to be a single rotation about the y-axis of 40° followed by a single rotation about the x-axis of 83°. Note that this isn't the same as rotating about the x-axis first and then the y-axis second which gives a different result. The fact that the order is important is a peculiar property of rotations. The fact that rotations can be written as matrices means that the order of multiplication is also a property of matrices.

Doing a rotation about the y axis first, followed by a rotation about the x axis, does provide a recipe for always getting to the same orientation every time. This is just like finding a position on the globe uniquely using circles of longitude and

latitude. The first rotation about the y axis gives the angle of latitude, and the second rotation about the x-axis gives the angle of longitude. This results in a simple scheme to orientate an object but, as we will see, the joystick response seems strange since what happens on the screen depends on the total current angles of rotation.

If this seems confusing then consider the complementary scheme of leaving the object stationary and moving the observer to different orientations at some fixed distance from the object. This is what has been done in the example program in this chapter. Figure 10.1 illustrates what is going on in the world frame. You can imagine a long pole, AB, between the object and the observer, with the observer looking down the pole towards the object. The rotations which take place change the orientation of the pole. In the example program, movement of the joystick left or right changes θ and movement up or down changes ϕ. We are now dealing with things the other way round to just rotating the object.

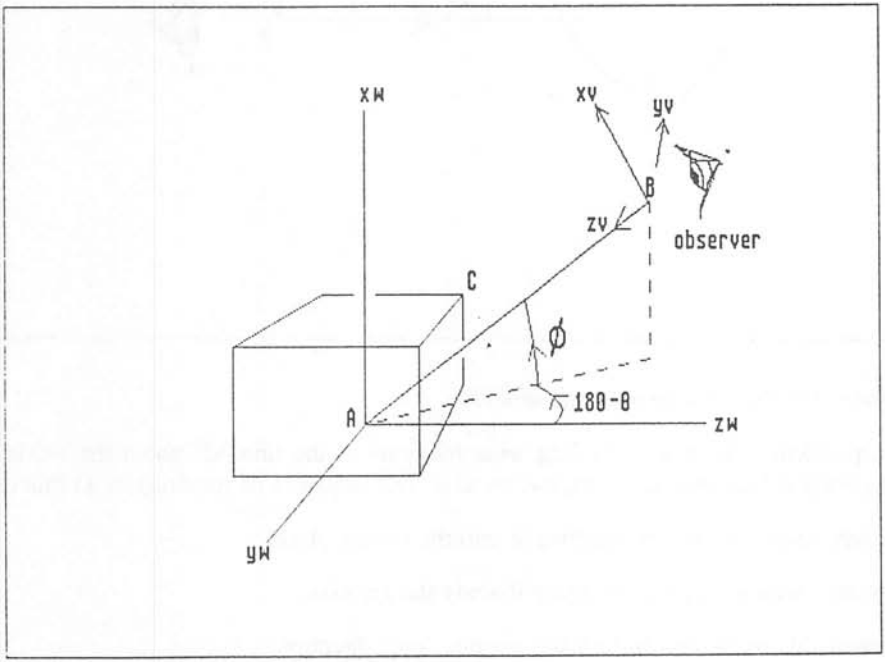

Figure 10.1 Rotating the observer about the object

The observer is at the angles shown in the figure and we have to find out what he/she sees. As drawn, the observer is closest to the vertex C and sees it pretty well head-on, so in the observer's reference frame (where the pole is horizontal) things appear as in Figure 10.2. How can this view be constructed from knowing

only the angles θ and φ, and the distance AB? Like most problems involving rotations it is easier than it looks and has a lot to do with the complementary nature of geometric (moving the object) and coordinate (moving the observer) transforms, which are discussed extensively in Appendix 7.

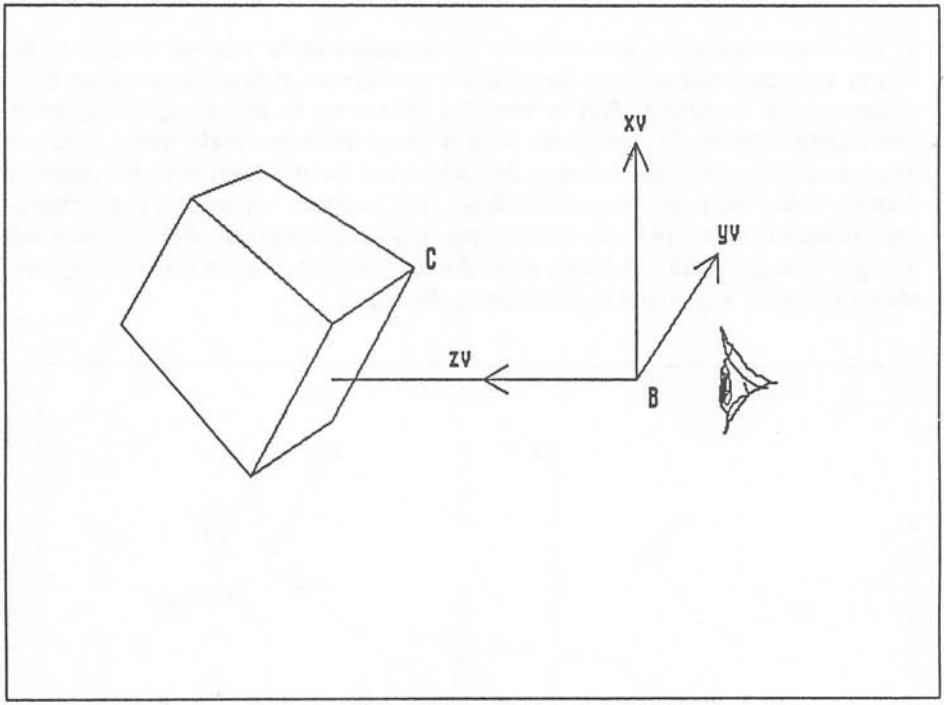

Figure 10.2 The view seen by the observer

The problem is solved by finding what rotations of the line AB about the world axes bring it back into line with the zw axis. The sequence of rotations to do this is

1. rotate about xw by (-θ) bringing it into the xw-zw plane,

2. rotate about yw by (-φ) bringing it along the zw axis,

(3. rotate about zw by (-γ) to make xw the "up" direction).

This last step is put in parentheses since it is not actually implemented in the program, i.e. there is no "twist" of the observer involved.

If this sequence of rotations is actually applied to the object with the viewer fixed in position along the world frame zw axis, then the overall result is the same. This is precisely what is done in the example program. The sequence of rotations which

must be applied to object about its centre are, in order (remember the one at the right acts first):

$$\begin{pmatrix} \cos\gamma & \sin\gamma & 0 \\ -\sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\phi & 0 & -\sin\phi \\ 0 & 1 & 0 \\ \sin\phi & 0 & \cos\phi \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{pmatrix}$$

which when multiplied (concatenated) out give the single matrix whose elements appear in the program. After transforming all the vertices with this matrix, all that remains to do is to add on the distance AB (also called Ovz) to each z coordinate.

We will use this particularly simple transform to the observer's reference frame again in Chapter 12 in a flight simulator where the angles (called Euler angles) can be easily related to joystick movement. It's OK if you don't mind the restriction of the way the angles are defined. In general, more freedom may be desired.

10.1.2 Scaling

Scaling is very straightforward. It simply makes the object larger or smaller. The scale change occurs independently along the three axes. For a general scale change, with different scale factors, a, b and c, along the three axes the transformation matrix is

$$\begin{pmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{pmatrix}$$

If both b and c are unity and a is greater than unity, then the resulting distortion is a stretch along the x axis. This is what is implemented in the example program. It is shown in Figure 10.3.

10.1.3 Shear

A shear distortion has the effect of displacing one face relative to its opposite. In the simplest case, one of the coordinates is increased in proportion to one of the others. If x increases in proportion to z, the matrix is:

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

and both y and z remain unchanged. This is illustrated in Figure 10.4 and included in the example program.

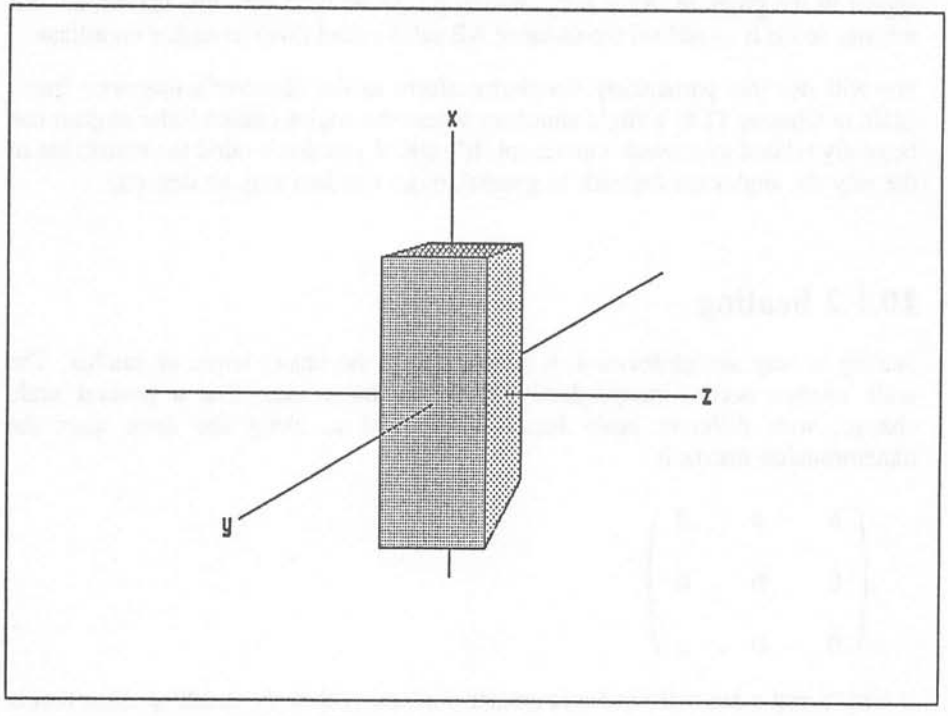

Figure 10.3 A stretch along the x axis

If x increases in proportion to both y and z the distortion becomes more exotic. This is shown in Figure 10.5 and also included in the example program. The matrix is

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

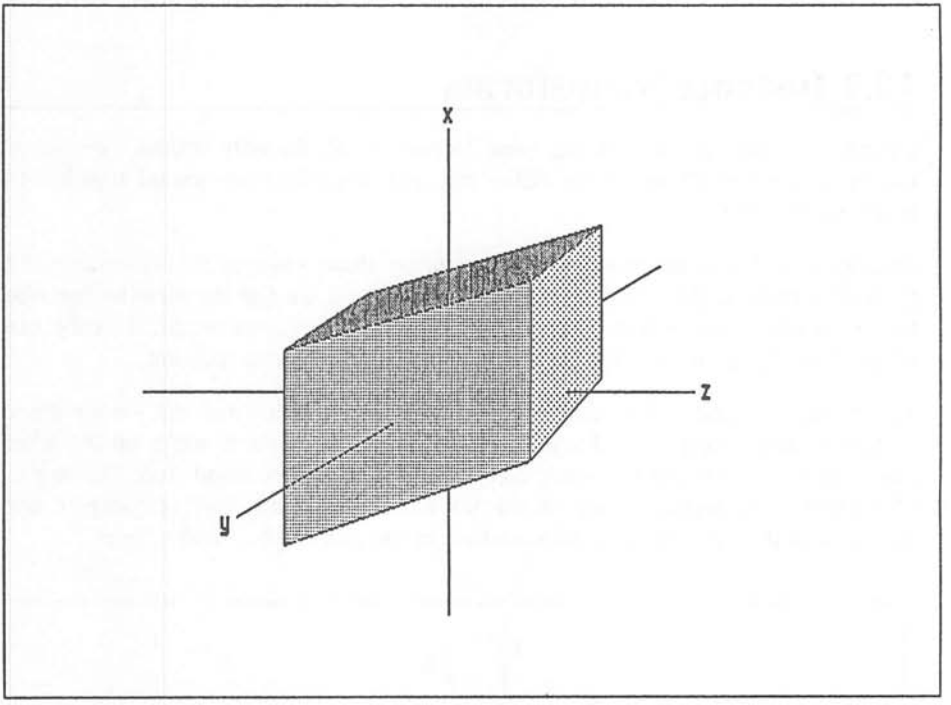

Figure 10.4 A shear in the x direction, proportioned to z

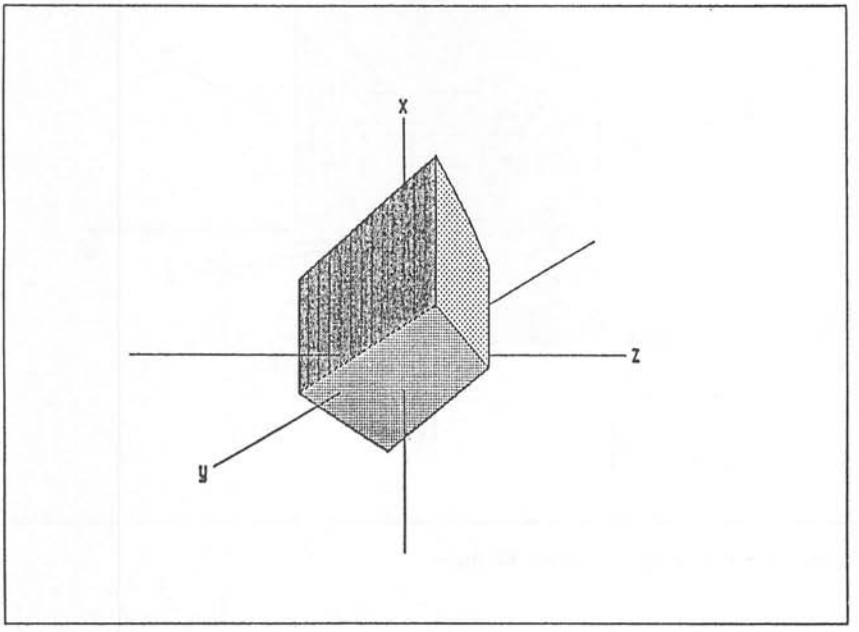

Figure 10.5 A sheer in the x direction proportioned to both y and z

10.2 Instance Transforms

Up till now, although motion has been 3-dimensional, the only structure displayed has been the flat ST monolith. Now, six such monoliths are joined together to make an ST cube.

Instance transforms are usually taken to mean those changes of orientation and position which set primitives in the world space and we use the term to describe the set of operations which construct the ST cube. Once constructed, the cube can be used as a basis to illustrate the transforms we have been discussing.

To construct a cube in this way, a monolith is first laid down in the yw-zw plane and then successively rotated and displaced five more times to make up the other sides. This is illustrated in Figure 10.6 where the sides are numbered. The angles of rotation and displacements of the six sides are in the lists *inst_angles* and *inst_disp* in the data file *data_05.s*, and are in the order θ, ϕ, δ and x, y, z.

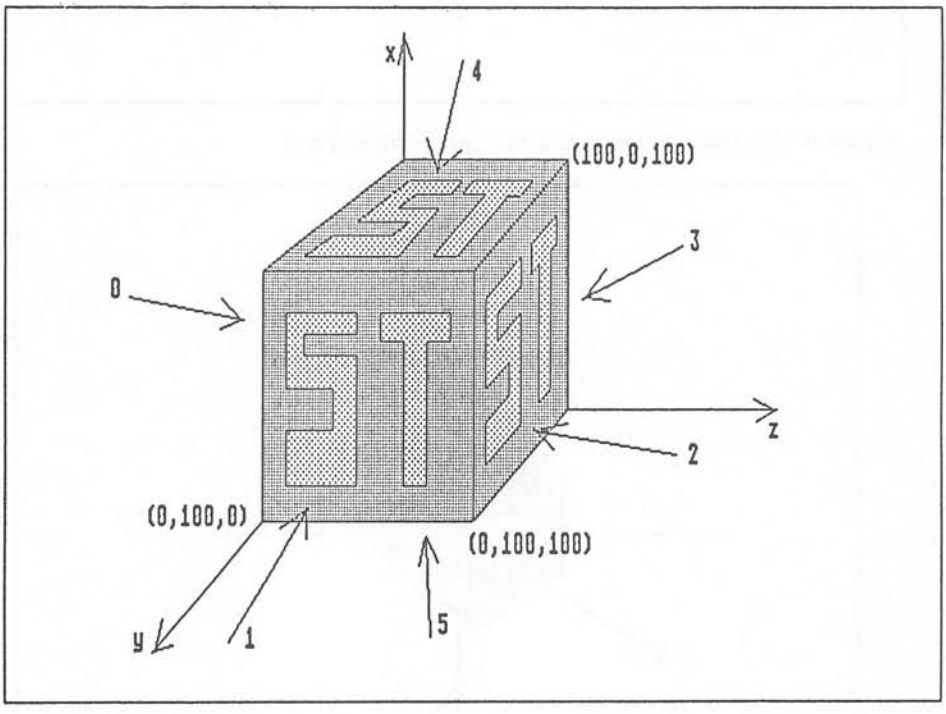

Figure 10.6 Construction of an ST cube

10.3 Physical Realism

Physical objects have more subtle attributes than shape and colour. This is particularly evident when motion occurs. Real objects do not move instantaneously from one place to another, nor do they achieve their final velocity the instant motion begins. There is an acceleration period whilst the velocity builds up to its maximum value. Likewise a real object cannot reduce its speed to zero instantaneously. A period of deceleration is required. Acceleration and deceleration are both evidence of an additional attribute of a physical object, its inertia or mass. The mass of an object determines how rapidly it can be accelerated or brought to rest. In building realistic computer models of physical objects it is important to pay attention to these details. The role of the mass of a body in determining its motion is really summarised in Newton's Laws of Motion. In essence, they say that if a body is acted on by a force it will accelerate in proportion to the force and, if there is no force, it remains at constant velocity (or at rest).

In the example programs, some attempt has been made to incorporate these laws by modelling joystick movements as applied forces. The result is that motion of the image does not follow immediately, but with an acceleration determined by its inertia. In addition, the effect of friction is incorporated so that if the applied force is removed the velocity drops to zero, and even when it is constantly applied there is a maximum to the velocity. In the programs, the motion is purely rotational but the same principles hold true.

10.4 Example Program

The program shows a cube with the letters ST written on each face in rotation under the control of the joystick. In addition the cube can be subject to shear and scaling transforms whilst the rotation takes place.

10.4.1 *trnsfrms.s*

This is the control program. After initializing variables, it reads the joystick and keyboard settings to choose the rate of rotation, viewing distance and whether a shear or scale change should take place. Both of these latter transforms are accompanied by a size reduction to keep word-size variables within range.

Once input is complete the cube is assembled, unrotated or distorted, in the world frame by the multiple object-to-world transform for all the sides. Following this the distortion is concatenated with the viewing transform to produce the overall transform which then converts the vertices for perspective projection.

10.4.2 *core_05.s*

Here are the new subroutines. The first part is concerned with constructing the rotation transform from the viewing angles vθ, vφ and vγ and then using it (after it is concatenated with the shear) to transform the vertices. Following this the routines are concerned with reading the joystick and keyboard and making adjustments accordingly.

In order to simulate inertia, movements of the joystick are converted not to angles of rotation themselves but as increments to the angles of rotation, up to a maximum. These increments are added to the angles each time to give the total angles to rotate. In addition, the increments are always decremented by 1 each time to give built-in frictional slowing down. The procedure to implement joystick alternatives uses a vector jump table to the various possible subroutines. This is an elegant way of avoiding testing for each possibility in a long list. This technique is also used for keyboard input.

There are seven possible keyboard inputs concerned entirely with the function keys f1 to f7:

f1- move closer (continuously) to a minimum distance,

f2- move away (continuously),

f3- implement shear 1 (x increases with z, called *xshear*),

f4- implement shear 2 (x increases with y and z, called *yshear*),

f5- implement a stretch (y and z reduced by 1/2),

f6- stop movement (of f1 and f2),

f7- quit.

Input from f3, f4 and f5 is used to set the bottom three bits of a word length flag, *shearflg*, in a toggle fashion using the bit-change instruction. This simply *NOT*'s the appropriate bit to provide a record of whether the transform should be implemented. The routine which examines which flag bits are set also includes the option of combinations of them which are not actually used for anything. They are called *user1* to *user4* and can be used to try other transforms (providing products do not exceed word size in the concatenation products).

Finally the shear and rotation matrices are multiplied to produce the overall transform to act on the cube.

10.4.3 *bss_05.s*

New variables for this chapter.

10.4.4 *data_05.s*

New data for this chapter. In particular note that the 3x3 matrices for the shears and stretch are arranged in column order to simplify the matrix concatenation routine.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                        trnsfrms.s                                     *
*                    Various 3D transforms                              *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

        SECTION TEXT
        opt     d+
        bra main
        include systm_02.s      screens and tables
        include systm_03.s      joystick
        include systm_04.s      set up screens, palette, joystick
        include core_05.s       motion of the view frame
*********************************************************************
main
        bsr     set_up          set up screens etc
* transfer all the data
        bsr     transfer
        move.w  oncoords,vncoords
        move.w  vncoords,wncoords
* Initialise dynamical variables
        move.w  #-50,Ovx        view frame initial position
        move.w  #0,Ovy
        move.w  #150,Ovz
        clr.w   vtheta          initialize rotation angles to zero
        clr.w   vphi
        clr.w   vgamma
        clr.w   shearflg        set flag to no shear
        move.w  #25,vtheta_inc  initial rotation rates
        move.w  #25,vphi_inc
        clr.w   speed
        clr.w   screenflag      0=screen 1 draw, 1=screen 2 draw
        bsr     clear1          clear the screens
        bsr     clear2
loop4:
* Switch the screens each time round
        tst.w   screenflag      screen 1 or screen2?
        beq     screen_1        draw on screen 1, display screen2
        bsr     draw2_disp1     draw on screen 2, display screen1
        bsr     clear2          but first wipe it clean
        clr.w   screenflag      and set the flag for next time
        bra     screen_2
screen_1:
        bsr     draw1_disp2     draw on 1, display 2
        bsr     clear1          but first wipe it clean
        move.w  #1,screenflag   and set the flag for next time
screen_2:
* look for changes in the rotation angles
        bsr     joy_in
* see if the function keys have been pressed to change the speed
* or initiate a shear
        bsr     key_in
* Adjust to new rotation angles and speed
        bsr     angle_update
        bsr     speed_adj
* Construct the compound object from the same face at different positions
        move.w  nparts,d7       how many parts in the object
        subq    #1,d7
        lea     inst_angles,a0  list of instance angles for each part
        lea     inst_disp,a1    ditto       displacements
```

```
* Do one face at a time
instance:
        move.w  d7,-(sp)        save the count
        move.w  (a0)+,otheta    next otheta
        move.w  (a0)+,ophi      next ophi
        move.w  (a0)+,ogamma    next ogamma
        move.w  (a1)+,Oox       next displacements
        move.w  (a1)+,Ooy
        move.w  (a1)+,Ooz
        movem.l a0/a1,-(sp)     save position in list
        bsr     otranw          object to world transform
        bsr     wtranv_1        construct the rotation transform
        bsr     shear           concatenate with shear (if flag set)
        bsr     wtranv_2        and transform the points
        bsr     illuminate      if it's visible find the shade
        bsr     perspective     perspective
        bsr     polydraw        draw this face
        movem.l (sp)+,a0/a1     restore pointers
        move.w  (sp)+,d7        restore the parts count
        dbra    d7,instance     for all the parts of the object
        bra     loop4           draw the next frame
        SECTION DATA
        include data_03.s
        include data_05.s
        SECTION BSS
        include bss_05.s

        END
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* Copyright A.Tyler 1991 systm_04.s                                     *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
set_up  bsr     find_screens    find the addresses of the two screens
        bsr     wrt_scrn1_tbl   write a row address table for screen1
        bsr     wrt_scrn2_tbl   ditto                        screen2
        bsr     hline_1u
        bsr     hide_mse        exterminate the mouse
        bsr     palette_set     set up the shades of blue and red
        bsr     init_joy        insert my joystick handler
        rts
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                        core_05.s                                      *
*                Subroutines for Chapter 10                             *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        include core_04.s               previous subroutines
        include core_03.s

* A set of subroutines for transforming world coords. including
* rotations of vtheta, vphi and vgamma about the x, y and z axes
* and x, y and z shears.

* The matrix for the rotations is constructed.
* First convert rotation angles to sin & cos and store for rot. matrix
wtranv_1:
        bsr     view_trig       find the sines and cosines
* Construct the transform matrix wtranv remember, all elements end up *2^14
        lea     stheta,a0       sin theta
        lea     ctheta,a1       cos theta
        lea     sphi,a2         sin phi
        lea     cphi,a3         cos phi
        lea     sgamma,a4       sin gamma
        lea     cgamma,a5       cos gamma
        lea     w_vmatx,a6      the matrix
* do element WM11
        move.w  (a3),d0         cphi
        muls    (a5),d0         cphi x cgamma
        lsl.l   #2,d0
        swap    d0              /2^14
        move.w  d0,(a6)+        WM11
* do WM12
        move.w  (a1),d0         ctheta
        muls    (a4),d0         ctheta x sgamma
        move.w  (a0),d1         stheta
        muls    (a2),d1         stheta x sphi
        lsl.l   #2,d1
        swap    d1
        muls    (a5),d1         stheta x sphi x cgamma
        add.l   d0,d1   stheta x sphi x cgamma + ctheta x sgamma
        lsl.l   #2,d1
        swap    d1
        move.w  d1,(a6)+
* do WM13
        move.w  (a0),d0         stheta
        muls    (a4),d0         stheta x sgamma
        move.w  (a1),d1         ctheta
        muls    (a2),d1         ctheta x sphi
        lsl.l   #2,d1
        swap    d1
        muls    (a5),d1         ctheta x sphi x cgamma
        sub.l   d1,d0   stheta x sgamma - ctheta x sphi x cgamma
        lsl.l   #2,d0
        swap    d0
        move.w  d0,(a6)+
* do WM21
        move.w  (a3),d0         cphi
        muls    (a4),d0         ctheta x sgamma
        lsl.l   #2,d0
        swap    d0              /2^14
        neg     d0
        move.w  d0,(a6)+
```

```
* do WM22
        move.w  (a1),d0         ctheta
        muls    (a5),d0         ctheta x cgamma
        move.w  (a0),d1         stheta
        muls    (a2),d1         stheta x sphi
        lsl.l   #2,d1
        swap    d1
        muls    (a4),d1         stheta x sphi x sgamma
        sub.l   d1,d0   ctheta x cgamma - stheta x sphi x sgamma
        lsl.l   #2,d0
        swap    d0
        move.w  d0,(a6)+
* do WM23
        move.w  (a0),d0         stheta
        muls    (a5),d0         stheta x cgamma
        move.w  (a1),d1         ctheta
        muls    (a2),d1         ctheta x sphi
        lsl.l   #2,d1
        swap    d1
        muls    (a4),d1         ctheta x sphi x sgamma
        add.l   d0,d1
        lsl.l   #2,d1
        swap    d1
        move.w  d1,(a6)+                " +stheta x cgamma
* do WM31
        move.w  (a2),(a6)+
* do WM32
        move.w  (a3),d0         cphi
        muls    (a0),d0         cphi*stheta
        lsl.l   #2,d0
        swap    d0              /2^14
        neg     d0              -
        move.w  d0,(a6)+
* d0 WM33
        move.w  (a1),d0         ctheta
        muls    (a3),d0         ctheta x cphi
        lsl.l   #2,d0
        swap    d0
        move.w  d0,(a6)+
        rts
* PART 2
* Now the world coords are transformed to view coords
* Remember matrix elements are *2^14 and must be corrected at the end
wtranv_2:
        move.w  wncoords,d7     the number
        ext.l   d7              any to do ?
        beq     wtranv3         if not quit
        subq.w  #1,d7           or this is the count

        lea     wcoordsx,a0     the
        lea     wcoordsy,a1     source
        lea     wcoordsz,a2     coords.
        lea     vcoordsx,a3     the
        lea     vcoordsy,a4     destination
        lea     vcoordsz,a5
        exg     a3,d3           save this address-short of regs.
        link    a6,#-6          3 words to store
```

```
wtranv1:
        moveq.l #2,d6           3 rows in the matrix
        lea     w_vmatx,a3      init matx pointer
* calculate the next wx, wy and wz
wtranv2:
        move.w  (a0),d0         wx
        move.w  (a1),d1         wy
        move.w  (a2),d2         wz

        sub.w   #50,d0          wx-50
        sub.w   #50,d1          wy-50
        sub.w   #50,d2          wz-50

        muls    (a3)+,d0        wx*Mi1
        muls    (a3)+,d1        wy*Mi2
        muls    (a3)+,d2        wz*Mi3

        add.l   d1,d0
        add.l   d2,d0           wx*Mi1+wy*Mi2+wz*Mi3
        lsl.l   #2,d0
        swap    d0              /2^14
        move.w  d0,-(a6)        save it
        dbf     d6,wtranv2      repeat for 3 elements

        move.w  (a6)+,d0        off my stack
        add.w   Ovz,d0
        move.w  d0,(a5)+        becomes vz
        move.w  (a6)+,(a4)+
        exg     a3,d3           restore address vx, save matx pointr
        move.w  (a6)+,d0
        add.w   #100,d0
        move.w  d0,(a3)+        becomes vx
        exg     a3,d3           save address vx, restore matx pointr
        addq.l  #2,a0           point to next wx
        addq.l  #2,a1                       wy
        addq.l  #2,a2                       wz

        dbf     d7,wtranv1      repeat for all ocoords
        unlk    a6              close frame
wtranv3 rts                     and quit
* Calculate the sines and cosines of view angles
view_trig:
        move.w  vtheta,d1       theta
        bsr     sincos
        move.w  d2,stheta       sine
        move.w  d3,ctheta       cosine
        move.w  vphi,d1         phi
        bsr     sincos
        move.w  d2,sphi
        move.w  d3,cphi
        move.w  vgamma,d1       gamma
        bsr     sincos
        move.w  d2,sgamma
        move.w  d3,cgamma
        rts
```

```
joy_in:
* Read the joystick and update the variables accordingly
* The data packet containing the FIRE bit (7) and the position
* bits (0-2) is saved in the variable joy_data
        clr.w   joy_data
        move.w  #10,d6
joy_again:
        bsr     rd_joy          read joystick
        dbf     d6,joy_again    give it time to think
* convert the joystick reading to angle totals
angle_speed:
        move.w  joy_data,d0     here's the result
        move    d0,d1           save it
        andi.w  #$f0,d0         fire pressed  ?
        bne     fire_press      yes
        andi.w  #$f,d1          what direction is the stick?
        bne     joy_dir
        rts
joy_dir lea     jump_table,a0   base address
        lsl.w   #2,d1           offset into jump table
        move.l  0(a0,d1.w),a0   the jump address
        jmp     (a0)            go for it
jump_table:
        dc.l    0,up,down,0,left,up_left,down_left
        dc.l    0,right,up_right,down_right
* set up the increments to angles - +-10 is the limit
up      subq.w  #2,vphi_inc
        rts
down    addq.w  #2,vphi_inc
        rts
left    addq.w  #2,vtheta_inc
        rts
right   subq.w  #2,vtheta_inc
        rts
up_left:
        addq.w  #2,vtheta_inc
        subq.w  #2,vphi_inc
        rts
down_left:
        addq.w  #2,vtheta_inc
        addq.w  #2,vphi_inc
        rts
up_right:
        subq.w  #2,vtheta_inc
        subq.w  #2,vphi_inc
        rts
down_right:
        subq.w  #2,vtheta_inc
        addq.w  #2,vphi_inc
        rts
fire_press:
        move.w  #1,fire
        rts
```

```
angle_update:
*Check the limits
        move.w  vtheta_inc,d0
        bmi     vth_neg
        beq     chk_phi
        subq.w  #1,vtheta_inc
        cmp.w   #25,vtheta_inc
        ble     chk_phi
        move.w  #25,vtheta_inc
        bra     chk_phi
vth_neg addq.w  #1,vtheta_inc
        cmp.w   #-25,vtheta_inc
        bge     chk_phi
        move.w  #-25,vtheta_inc
chk_phi move.w  vphi_inc,d0
        bmi     vph_neg
        beq     chk_out
        subq.w  #1,vphi_inc
        cmp.w   #25,vphi_inc
        ble     chk_out
        move.w  #25,vphi_inc
        bra     chk_out
vph_neg addq.w  #1,vphi_inc
        cmp.w   #-25,vphi_inc
        bge     chk_out
        move.w  #-25,vphi_inc
chk_out


* update vtheta
        move.w  vtheta,d0       the previous angle
        add.w   vtheta_inc,d0   increase it by the increment
        bgt     thta_1          check it
        add     #360,d0         lies
        bra     thta_2          between zero
thta_1  cmp.w   #360,d0         and 360 degrees
        blt     thta_2
        sub     #360,d0
thta_2:
        move.w  d0,vtheta       becomes the current angle
* update vphi
        move.w  vphi,d0         similarly for vphi
        add.w   vphi_inc,d0
        bgt     phi_1
        add     #360,d0
        bra     phi_2
phi_1   cmp.w   #360,d0
        blt     phi_2
        sub     #360,d0
phi_2:
        move.w  d0,vphi
        rts

key_in:
* Read the keyboard
        bsr     scan_keys       was a key pressed?
        cmp.w   #-1,d0
        beq     key_read        yes
        rts                     no
```

```
key_read:
        bsr     read_key        which key?
        tst.w   d0              f keys have $0 in the low word
        beq     key_rpt         only interested if f keys
        rts                     something else
key_rpt swap    d0              the code
        sub     #$3b,d0         f1 is 3b : set it to zero for offset
        and     #7,d0           first 7 f keys
        lea     key_jump,a0     jump table
        lsl.w   #2,d0           key code is offset
        move.l  0(a0,d0.w),a0   to the routine address
        jmp     (a0)            go for it
key_jump:
* The jump table for f keys
        dc.l    f1,f2,f3,f4,f5,f6,f7
f1      move.w  #-1,speed       reverse
        rts
f2      move.w  #1,speed        forward
        rts
f3      bchg.b  #2,shearflg     toggle x shear flag (reverse flag)
        rts
f4      bchg.b  #1,shearflg     toggle y shear flag
        rts
f5      bchg.b  #0,shearflg     toggle z shear flag
        rts
f6      move.w  #0,speed        stop
        rts
f7      clr.w   -(sp)
        trap    #1              quit altogether- return to caller

*Concatenate the shear with the rotation
shear   clr     d0
        move.b  shearflg,d0     the shear flags are the 3 low bits
        and     #$f,d0
* there are 8 possibilities: 111,110,101,100,011,010,001,000
* the bit numbers refer to x, y and z shears respectively
        lea     shear_jump,a0   jump table base
        lsl.w   #2,d0           shear code is offset
        move.l  0(a0,d0.w),a0   to routine address
        jmp     (a0)            go for it
shear_jump:
* The jump table
        dc.l    null,z,y,user1,x,user2,user3,user4
null    rts     do nothing
z       lea     zshear,a0       pointer to shear
        lea     w_vmatx,a1      pointer to rotation
        bsr     concat          concatenate them
        rts
y       lea     yshear,a0
        lea     w_vmatx,a1
        bsr     concat
        rts
user1   rts
x       lea     xshear,a0
        lea     w_vmatx,a1
        bsr     concat
        rts
user2   rts
user3   rts
user4   rts
```

```
concat:
* Concatenate (multiply) the two 3x3 matrices pointed to by a0 and a1.
* The order is (a1)x(a0) with the result sent to temporary store at (a2).
* (a0) is in column order and (a1) and (a2) are in row order,
* of word length elements. Finally (a2) is transferred to (a1)
        lea     tempmatx,a2     temporary store
        moveq.w #2,d7           3 rows in a1
conc1   move.w  #2,d6           3 columns in a0 (shear)
        movea.l a0,a3           reset shear pointer
conc2   move.w  (a1),d1         next rot. element
        ext.l   d1
        lsr.l   #1,d1           /2
        move.w  2(a1),d2
        ext.l   d2
        lsr.l   #1,d2
        move.w  4(a1),d3
        ext.l   d3
        lsr.l   #1,d3
        muls    (a3)+,d1
        muls    (a3)+,d2
        muls    (a3)+,d3
        add.w   d2,d1
        add.w   d3,d1
        move.w  d1,(a2)+        next product element
        dbra    d6,conc2        for all elements in this row
        addq.w  #6,a1           pointer to next row
        dbra    d7,conc1        for all rows
* transfer the result back to the rotation matrix
        lea     tempmatx,a0     temp. store of product
        lea     w_vmatx,a1      becomes new transform
        moveq.w #8,d7           9-1 elements in 3x3 matx
conloop move.w  (a0)+,(a1)+     next element
        dbra    d7,conloop      for all elements
        rts
* Set the velocity components
speed_adj:
        move.w  speed,d0
        lsl.w   #3,d0           scale it
        move.w  Ovz,d1
        cmp.w   #10,Ovz         musn't come any closer
        bgt     adj_out
        move.w  #10,Ovz
adj_out add.w   d0,Ovz          zw speed component
        rts
```

```
*                       bss_05.s
* Additional variables for Chapter 10
        include bss_04.s        surface illumination vars.
        include bss_03.s
* world frame variable
wncoords        ds.w    1       no. of vertices in world frame
*view frame variables
vtheta          ds.w    1       rotation of view frame about wx
vphi            ds.w    1       ditto                        wy
vgamma          ds.w    1       ditto                        wz
Ovx             ds.w    1       view frame origin x  in world frame
Ovy             ds.w    1       ditto             y
Ovz             ds.w    1       ditto             z
* The general transform matrices
w_vmatx         ds.w    9       the matrix elements
tempmatx        ds.w    9       temporary store for matx products
* Variables for the intelligent keyboard
gem_joy         ds.l    1       store gem joystick handler
joy_data        ds.w    1       jostick direction/fire
gem_mse         ds.l    1       store gem mouse handler
mse_click       ds.w    1       click flag
mouse_dx        ds.w    1       x displacement since last
mouse_dy        ds.w    1       y ditto
* Dynamical variables
speed           ds.w    1
vtheta_inc      ds.w    1
vphi_inc        ds.w    1
vgamma_inc      ds.w    1
fire            ds.w    1       mse fire flag
shearflg        ds.w    1       shear flags
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                        data_05.s                                          *
*                                                                           *
*        Data for Chapter 10. An ST cube                                    *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

my_datax        dc.w    100,100,0,0,85,85,15,15,75,75,65,65,50,50,40,40
                dc.w    85,85,70,70,70,70,15,15
my_datay        dc.w    0,100,100,0,10,40,40,10,20,40,40,20
                dc.w    10,30,30,10,50,90,90,50,60,80,80,60
my_dataz        dc.w    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
my_edglst       dc.w    0,1,2,3,0,4,5,6,7,4,8,9,10,11,8,12,13,14,15,12
                dc.w    16,17,18,19,16,20,21,22,23,20
my_nedges       dc.w    4,4,4,4,4,4
my_npoly        dc.w    6
nparts          dc.w    6
inst_angles     dc.w    0,0,0,90,0,0,180,0,0,270,0,0,0,270,0,0,90,0
inst_disp       dc.w    0,0,0,0,100,0,0,100,100,0,0,100,100,0,0,0,0,100
my_xmin         dc.w    0
my_xmax         dc.w    319
my_ymin         dc.w    0
my_ymax         dc.w    199

ill_vecx        dc.w    0
ill_vecy        dc.w    -16384  light shining from +y to -y
ill_vecz        dc.w    0
vwpointz        dc.w    -100
illkey          dc.w    1
intr_col        dc.w    0,1,0,0,1,1
xshear          dc.w    1,0,0,0,1,0,1,0,1       1 shear
yshear          dc.w    1,0,0,1,1,0,1,0,1       2 shear
zshear          dc.w    2,0,0,0,1,0,0,0,1       3 stretch
palette dc.w    0,$557,$446,$336,$226,$225,$114,$113
        dc.w    $756,$745,$734,$723,$713,$702,$502,$401
```

11

Flying Around The World

11.1 Introduction

A flight simulator? Well not exactly, but getting there.

In order to fully implement the simulation of independent motion of the observer, we require a little more vector algebra. The task is to construct a view of the world model from the point of view of an observer free to move in any direction. This is different from the simple procedure we used in the previous chapter. We now wish to operate a joystick and navigate our way through the assembly of objects constructed in the world frame. We want the view on the screen to move up or down when the joystick is pushed forward or pulled back and to move to the left or right when the joystick is moved to the right or the left. In other words, all of the motion on the screen must be relative to the observer's current position. Even if the pilot of a plane is flying upside down, his perception of ''up'' is directed towards the roof of the cockpit, which as far as someone on the ground is concerned is ''down''. What matters is that all of the movements corresponding to ''up'', ''down'', ''left'' and ''right'' apply to the observer's reference frame, which we have called the view frame. Unlike rotation by Euler angles, which we used in the previous chapter, here we want the rotations to be about the view frame axes.

To be specific, let's ask what we expect to happen when the joystick is pulled back. We expect to see the picture move vertically upwards, and this must always happen no matter what the orientation of the observer. Suppose we have got into the position where the aircraft, or whatever it is being controlled, is flying horizontally but with its wings vertically. Figure 11.1 shows this orientation. If the joystick is pulled back, object A will come into view at the top of the screen and the object B will go out of view at the bottom of the screen. The view seen by the

pilot of the plane is shown in Figure 11.2. Herein lies the problem. The pilot has a very definite perception of what is "up" and what is "down" at any given moment and, while this does not change in the cockpit, it is changing continuously with respect to the world outside. In the previous chapter it was easy to relate "up" to an increase of $v\phi$ and left to an increase in $v\theta$, but when referenced from the view frame all these motions depend on the orientation of the observer at any given instant.

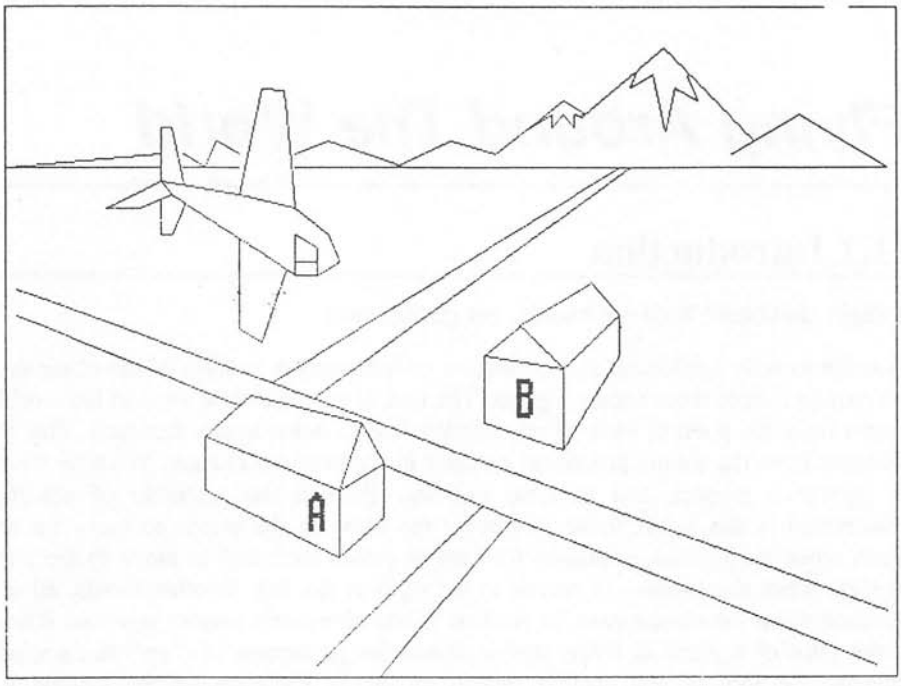

Figure 11.1 World view of observer (airplane)

There is more than one way of solving this problem. One method is to use control matrices to perform rotations of coordinates after they have been transformed to the view frame. The control matrices perform simple rotations about the view x, y and z axes. This method is employed in the next chapter. Another way is to keep a constant record of the position and orientation of the view frame in the world frame and to generate movements of the view frame resulting from movements of the joystick. This second method relies heavily on the notion of a set of view frame axes undergoing rotations and translations following the path of the observer. It also embodies the notion of rotation about an arbitrary axis that we would also like to introduce in this chapter which is very useful for performing rotations about any axis in the world frame.

We could, of course, decide to accept the limitations of Euler angles to fix the view frame orientation in the simpler orbital-like fashion. In Chapter 12, we show how a flight simulator works well by using each of these approaches.

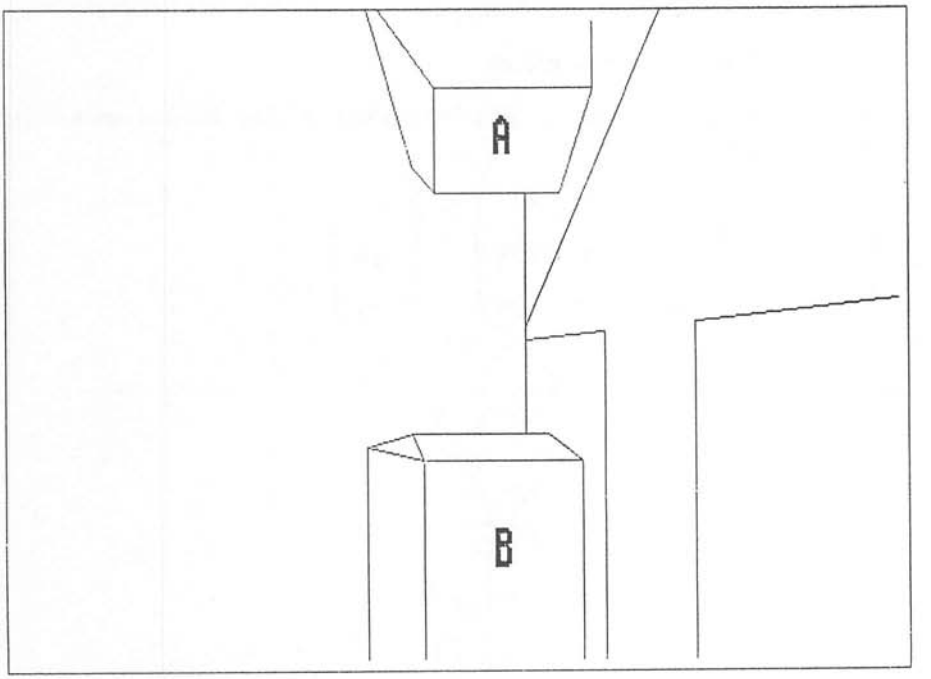

Figure 11.2 Observer's view (from airplane)

11.2 Coordinate Transforms and Direction Cosines

Here's a bit of maths. It's not as hard as it looks.

If you know the coordinates of the vertices of an object in one reference frame and want to know what they are in another, it is necessary to do a coordinate transform. (Remember the other type of transform is called a geometric transform, which is what happens when the object itself is moved inside a single reference frame). If a point has coordinates (xw,yw,zw) in the world frame, it will have coordinates (xv,yv,zv) in the view frame. Thus the point A in Figure 11.3 has coordinates (0,0,50) in the world frame, and coordinates (0,-50,0) in the view frame (what is seen on the screen has later to be worked out by means of the

perspective transform). As far as rotations are concerned there is always a linear relation between these two sets of coordinates, and for this case we can write in general terms:

$$xv = n11.xw + n12.yw + n13.zw$$

$$yv = n21.xw + n22.yw + n23.zw$$

$$zv = n31.xw + n32.yw + n33.zw$$

where the n's are numbers that remain to be worked out. This relation can also be written as a matrix product:

$$\begin{pmatrix} xv \\ yv \\ zv \end{pmatrix} = \begin{pmatrix} n11 & n12 & n13 \\ n21 & n22 & n23 \\ n31 & n32 & n33 \end{pmatrix} . \begin{pmatrix} xw \\ yw \\ zw \end{pmatrix}$$

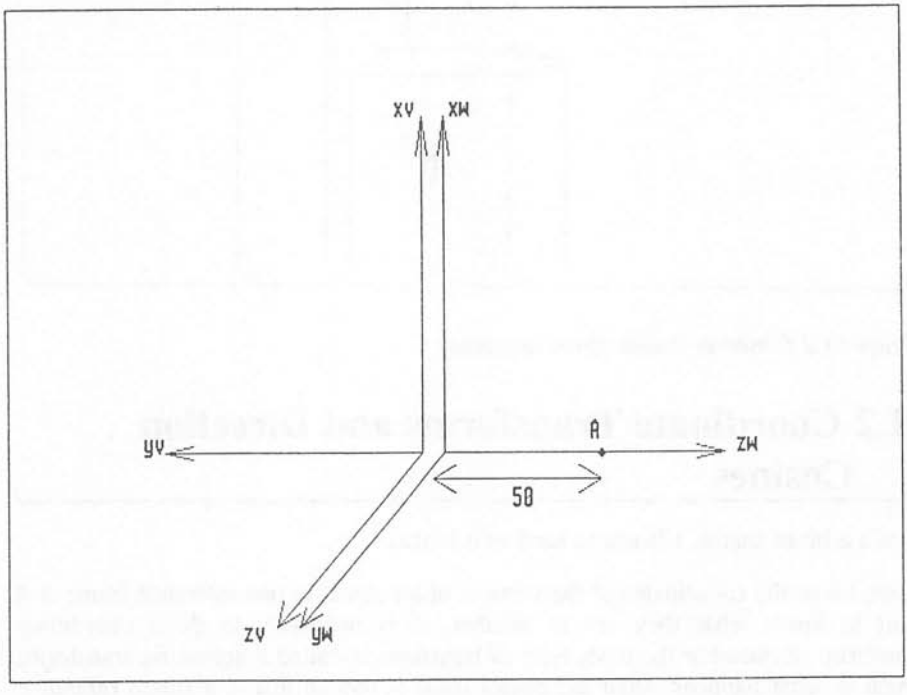

Figure 11.3 Point A seen in two coordinate frames

The n matrix is the transformation matrix. The elements n11, n12, etc., are specific to the relative orientation of the two reference frames and are called the direction cosines.

To see how the direction cosines are related to the geometry, look at Figure 11.4. The direction cosines are simply the cosines of the angles between the axes of the reference frames. It is quite hard to draw a comprehensive diagram which is not confusingly messy but, for example, n11 is the cosine of the angle between vx and wx, n12 is the cosine of the angle between vx and wy, n13 is the cosine of the angle between vx and wz and so on:

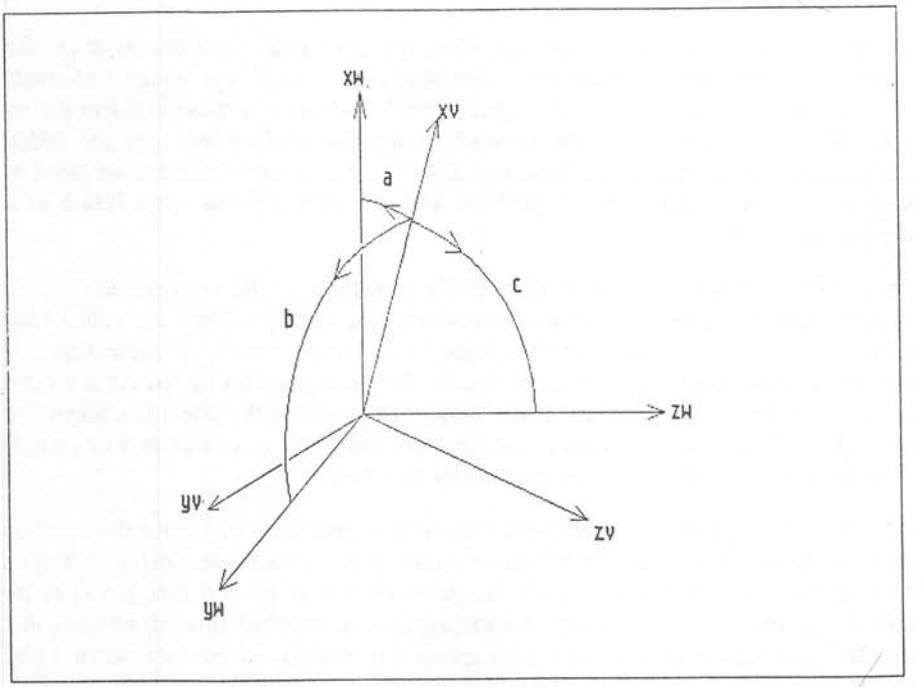

Figure 11.4 Direction cosines

$$n11 = \cos(a),\ n12 = \cos(b),\ n13 = \cos(c).$$

If these direction cosines can be found, the problem of converting world frame coordinates into view frame coordinates is solved. We are however still left with the problem of converting movements of the joystick into changes in the direction cosines. It is clear that we should solve the problem with a strategy that centres on the direction cosines. Here is one way it can be done.

11.3 Base Vectors and Direction Cosines

Just for a moment let's forget all about the maths. Let's try to visualise what's going from the point of view of a second, stationary, independent observer at rest in the world frame and able to see both the world frame and the moving observer simultaneously. This is the point of view of a man on the ground watching a plane fly past. Think of the plane as the view frame but with the fuselage replaced by the zv axis, the wings replaced by the yv axis and the vertical tail wing in the direction of the xv axis. Although he is not in the plane, the stationary observer can calculate the view according to the pilot if he knows the position and orientation of the plane at any instant.

To see how that view would change when the pilot pulls back the joystick, for example, he has only to rotate the plane about the axis of the wings (the angle depends on how long the joystick is pulled back), which is a rotation about the yv axis. Since the plane is moving forward during the rotation this has the added complication of making it fly upwards. Like the stationary observer, we need to keep a continuous record of the position and orientation of the view frame as it flies around the world.

To do this, imagine three unit vectors in the directions of the view frame axes. In vector geometry these unit vectors are given a special name. They are called base vectors. At the very start of the program let us suppose that the view frame is positioned coincident with the world frame. This is equivalent to having a second set of world frame base vectors at the airfield from where the plane has taken off. (Actually it isn't really necessary to have them start off coincident and in general they don't, but it makes the argument easier to visualise).

Now at each stage of the subsequent motion it is necessary to record the position and orientation of the view frame unit vectors. It is not possible simply to keep a running total of how many degrees the plane rotated to the left (about vx) or up (about vy) since we have no way of knowing how to translate this information into the final orientation of the plane after many movements. In the method of Euler angles used in the previous chapter it was possible to keep a running total since the first angle referred to rotation about an axis of the static world frame. But now we are using angles referred to the view frame which is moving all the time.

Here comes the big question. Suppose we can keep a record of the positions of the view frame base vectors, what do they have to do with the original transform? The answer is very simple: the components in the world frame of the view frame base vectors are just the direction cosines that are the elements, n11 to n33, of the world-to-view transform matrix. In other words, where **iv**, **jv** and **kv** are the view frame base vectors and **iw**, **jw** and **kw** are the world frame base vectors, the relation between them is:

$$\mathbf{iv} = n11.\mathbf{iw} + n12.\mathbf{jw} + n13.\mathbf{kw}$$

$$\mathbf{jv} = n21.\mathbf{iw} + n22.\mathbf{jw} + n23.\mathbf{kw}$$

$$\mathbf{kv} = n31.\mathbf{iw} + n32.\mathbf{jw} + n33.\mathbf{kw}.$$

Or, writing the view frame base vectors in terms of their world frame components

$$\mathbf{iv} = \begin{pmatrix} n11 \\ n12 \\ n13 \end{pmatrix}, \quad \mathbf{jv} = \begin{pmatrix} n21 \\ n22 \\ n23 \end{pmatrix}, \quad \mathbf{kv} = \begin{pmatrix} n31 \\ n32 \\ n33 \end{pmatrix}$$

At the start of the motion, when the view frame and world frame axes were aligned, the view frame base vectors had components

$$\mathbf{iv} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \quad \mathbf{jv} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad \mathbf{kv} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

If we can keep a record of the view frame base vectors we therefore have the direction cosines immediately available to construct the view from the cockpit. The strategy is straightforward but there are some tricky problems to solve on the way.

11.4 Rotating the Base Vectors: Rotation About an Arbitrary Axis

The base vectors which fix the current orientation of the view frame depend on what movements have already taken place. Suppose at a given instant the view frame is oriented with its base vectors in the positions shown in Figure 11.5. The base vector of the vx axis, iv, has three components in the world frame n11, n12 and n13 (the other unit vectors jv and kv also have components but for clarity these are not shown in the diagram). Now suppose a movement of the joystick occurs corresponding to a rotation about the vy axis. To find the new components of iv and kv (jv remains unchanged in this rotation) we must rotate them about vy. The vy axis is the axis of rotation and is specified in the world frame by its direction cosines. But we are in luck! This problem has already been solved. It is known as rotation about an arbitrary axis. Since at this point yv can be pointing anywhere in the world frame, the axis is very arbitrary. In fact the solution to the problem is given in just the format most useful to us. It is in the form of a matrix for rotation by an angle about an axis specified by its direction cosines. Just the form we want. The transform can also be used for rotation about any other axis in the world frame. All that is required are the three direction cosines.

Once constructed, the rotation matrix can be multiplied by **iv** and **kv** to yield the new components of **iv** and **kv**, which then replace the old ones and are also used directly to construct the world-to-view transform (there is a catch, which we'll discuss shortly).

For rotation by an angle δ about an axis with direction cosines n1, n2 and n3 (just the last index in the cosine to show it can refer to any axis), the matrix is

$$\begin{pmatrix} n1.n1+(1-n1.n1)\cos(\delta) & n1.n2(1-\cos(\delta))-n3\sin(\delta) & n1.n3(1-\cos(\delta))+n2\sin(\delta) \\ n1.n2(1-\cos(\delta))+n3\sin(\delta) & n2.n2+(1-n2.n2)\cos(\delta) & n2.n3(1-\cos(\delta))-n1\sin(\delta) \\ n1.n3(1-\cos(\delta))-n2\sin(\delta) & n2.n3(1-\cos(\delta))+n1\sin(\delta) & n3.n3+(1-n3.n3)\cos(\delta) \end{pmatrix}$$

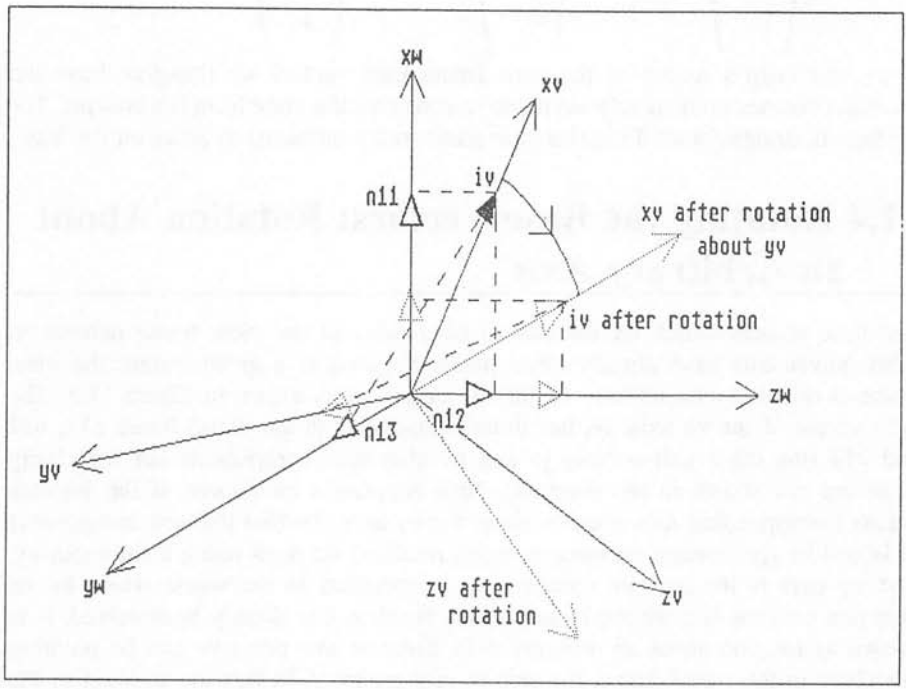

Figure 11.5 Rotation of base vector

11.5 Accumulating Errors

Broadly speaking, all the ingredients required to steer the view frame through the world frame controlled by joystick movements are in place. Let us lay out the algorithm as it stands at the moment:

- movement of the joystick specifies a rotation of the view unit vectors about one of the view frame axes,
- construct the rotation matrix to rotate the other two unit vectors about this axis and replace them with their new components,
- use the components of the unit vectors, now called direction cosines, to construct the world-to-view transform,
- perform the transform and display the picture
- and repeat the cycle.

This is all O.K. and it works. For a while.

Eventually it will lead to a degenerating picture, or worse a chaotic mess, because of accumulating errors. As it stands the program has a built-in pathological self-distruct. Because calculations are done in integer arithmetic, and sines and cosines are calculated to an accuracy no better than 1 in 16384, given enough transforms, large errors will accumulate in the unit vectors and, as a consequence, the world-to-view transform. In life nothing is perfect and this is a good example of that adage. In addition, the algorithm has feedback in that joystick movements are made on the basis of the picture on the screen that is generated, in turn, from the transform constructed from the joystick movements. This has all the ingredients necessary to create chaos, and so it does.

In order to beat the accumulation of errors, the cycle of error accumulation must be broken. This is achieved by regenerating the base vectors afresh each time. This requires more work but it solves the problem. Figure 11.6 shows the stages in the regeneration of the view frame unit vectors.

The vectors that matter most are **kv**, the one that points in the direction of motion and iv, the pointer to the "up" direction. Without these two it is not possible to define either the direction of motion or which way is up as far as the pilot is concerned. Let's suppose that, because of errors in the last transform, we have three unit vectors **iv'**, **jv'** and **kv'** which are slightly wrong. The errors will result in the base vectors not being at right-angles to each other and not having size equal to unity. As a first step, the vector **kv'** is normalised, i.e. its magnitude is made to be unity. It becomes **kv**. This at least ensures that if its direction is slightly wrong, its size isn't. The only effect a slightly wrong direction will have is that the view

will be slightly in error, but that hardly matters since the view is being constantly adjusted by the joystick anyway. Second, the vector cross product of **kv** and **iv'** is taken in order to generate a new vector at 90 degrees to them both. A vector cross product has just this property (see Appendix 6). This new vector is in the direction that **jv** would have if it weren't in error. The new vector is then normalised i.e. its magnitude is made to be 1, and it becomes the new **jv**. Third, the vector cross product of the new **kv** and the new **jv** is taken, and normalised, in order to generate a new **iv**. In this way all three unit vectors are regenerated each frame and errors do not accumulate (it is interesting to remove the regeneration stage in the example program and watch the disintegration take place).

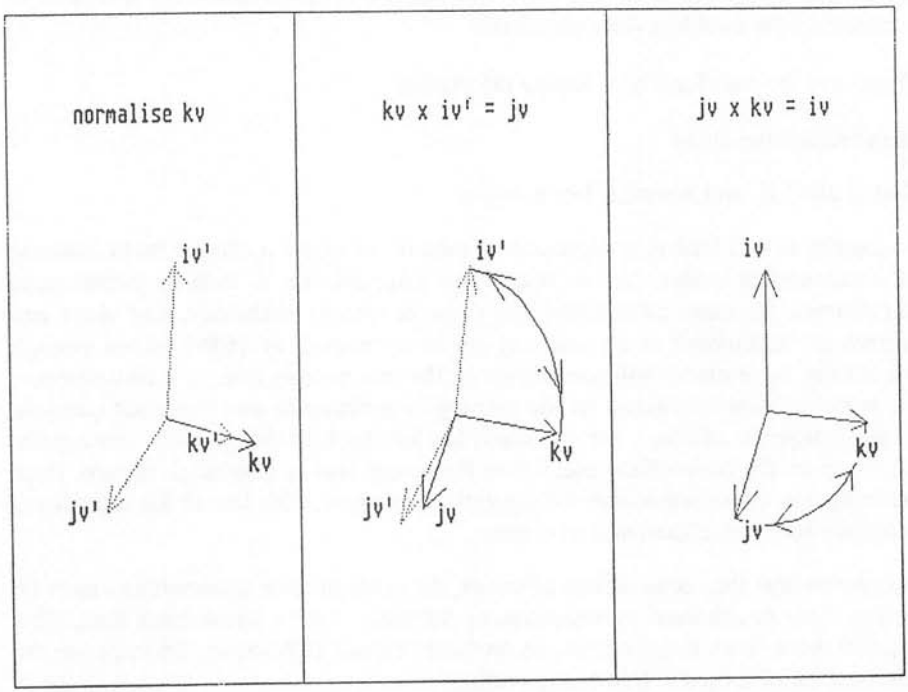

Figure 11.6 Regeneration of base vectors

The components of the new unit vectors then become the components of the viewing transform matrix and the cycle is repeated.

The technical details are discussed as they appear in the example programs.

11.6 Clipping in 3D

No part of an object which lies behind the view plane ($zv < 0$) must be drawn. If this is attempted, the program will not crash but what appears on the screen will be garbage. This is because the polygon drawing routines expect to see the edge list of vertices go clockwise round the perimeter of a polygon and this will be wrong for polygons projected backwards onto the view plane. In addition, objects that lie too far from the view plane should not be drawn either. This is because nothing can be drawn smaller than a pixel, and very distant objects reduce to an incoherent cluster of pixels.

Besides these obvious cases, there is no point in wasting time on objects that lie too far outside the field of view. This field of view is defined by the frustum (truncated pyramid) defined by the line of sight from the view point to the viewport boundaries. This is illustrated in Figure 11.7.

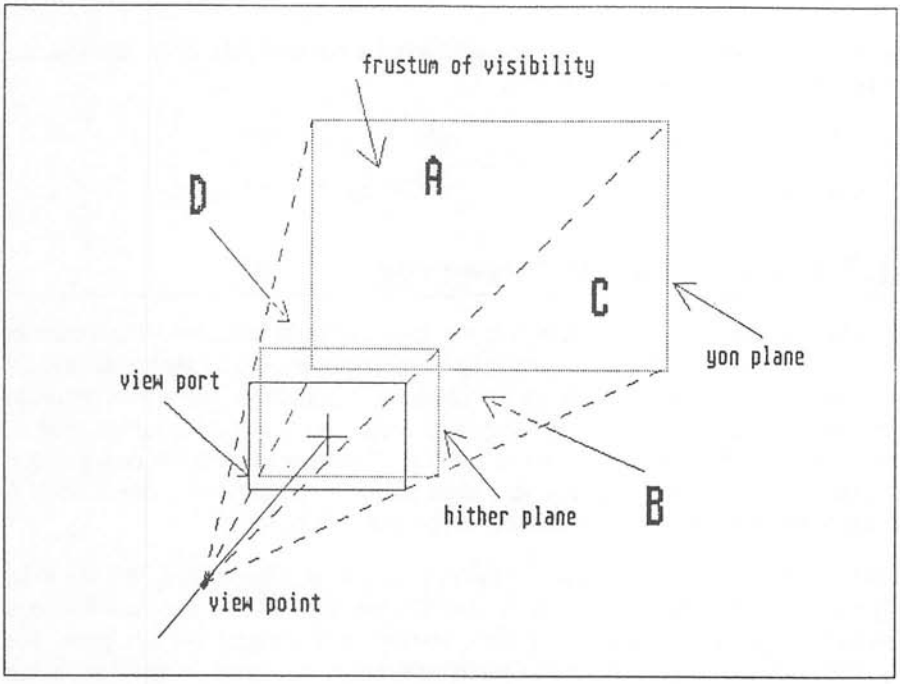

Figure 11.7 Windowing im 3D

In a more leisurely application it would be possible to clip polyhedra to the boundary of the frustum in a 3D generalisation of the way polygons have been clipped to the screen window. In this application that would be too time

consuming. Here, the centre of symmetry (Oox,Ooy,Ooz) is used to locate objects in the field of view and the angle of the frustum is increased to lie beyond the screen limit. This means that some time is wasted drawing distant objects which cannot be seen, but objects that are close up are not abandoned the instant their centres pass beyond the field of view. They are marked as visible but only part will appear on the screen as a result of screen clipping.

The top and base of the frustum are called the hither and yon planes. In the example program they are defined by the equations zv=100 (hither) and zv=2000 (yon). The sides of the frustum of the field of view are defined (where the viewport centre coincides with the view frame origin) by the planes

zv + 100 = xv	side A
zv + 100 = –xv	side B
(1.2).(zv + 100) = yv	side C
(1.2).(zv + 100) = –yv	side D

but the actual sides used in the program extend beyond this limit, for reasons explained above, and are described by

8*(zv + 100) = ±xv	sides A and B
8*(zv + 100) = ±yv	sides C and D

11.7 Velocity of the Observer

The observer (you) does not only use the joystick to do rotations. The observer also has a velocity that may be changing as time passes. To include velocity, all that has to be done is to increment the observer's position in the world frame in proportion to the velocity. The velocity is a vector, so it has direction as well as size - speed is the magnitude of the velocity. The procedure is to change each component of the observer's position, each frame, by an amount proportional to the speed times the relevant component of the base vector kv.

In other words, if the view frame is pointing only in the direction of the zw axis, only Ovz should be incremented each time. On the other hand if the view frame is pointing along the xw axis, only Ovx should be incremented each time. For anything between, Ovx, Ovy and Ovz should be incremented in proportion the components of kv in those directions. This ensures that the direction in which the observer is looking is the direction of motion. The details are explained in the example program.

11.8 Example Programs

In this program it is possible to fly round the ST cube. The program starts with the cube at mid-screen and with the observer stationary. Pressing f2 causes the view frame to move towards the cube at constant speed (pressing f1 causes it to retreat). Thereafter motion is controlled by the joystick. It is possible to fly past the cube and then do an about turn to return to it. Because of 3D clipping. the cube is not displayed if it comes closer than 100 or is farther away than 2000 or is outside the field of view (see above). Motion can be stopped by pressing f6 and the program aborted by pressing f7.

11.8.1 *wrld_vw.s*

This is the control program. Much of it is similar to that of the previous chapter. It draws an ST cube that can be flown around under the control of the joystick. This time the joystick performs rotations about the axes of the view frame, i.e. the pilot. When the joystick is pulled back the viewer looks upwards into the world and if there is forward motion he/she follows a rising trajectory. Other motions of the joystick produce corresponding motion as if the viewer were flying through the world frame. In this way it is possible to fly past an object and then sweep through an arc to return to it.

The program follows the sequence described above. First the view frame base vectors are initialized. Following this the joystick is read and immediately the view frame unit vectors in the world frame are rotated. Then the keyboard is read to see if the speed has changed. Following this the new position of the view frame in the world frame is calculated from the speed and the view frame z-axis base vector kv which is now pointing along the new direction of motion. In motion that is not in a straight line, the velocity is changing all the time (the velocity is a vector and so it can change if its direction changes even if its size, the speed, doesn't). Finally the unit vectors are themselves regenerated to avoid accumulating errors and passed on directly as the elements of the world-to-view transform before drawing the picture of the ST cube.

The function keys f1 and f2 are reverse and forward respectively. f6 is stop and f7 returns to the calling program. Be careful to press the keys lightly and not hold them down since the keyboard buffer is not cleared between frames.

There are no subtleties such as inertia in the motion but these could be incorporated along the lines described in the previous chapter.

11.8.2 *core_06.s*

Here is where all the work is done. The subroutine dircosines regenerates the base vectors and passes the new values to the viewing transform matrix. To do the

regeneration requires vector cross products and normalisation (i.e. scaling the size of the vector to unity). To normalise a vector requires dividing each of its components by the magnitude of the vector, which must be calculated as the square root of the sum of the squares of the components. This is dealt with using the *nrm_vec* routine used previously for the illumination calculation.

In the subroutine *in_joy*, the joystick is read and action taken immediately to rotate the view frame base vectors about an axis in the world frame, which here is one of the base vectors, but could be any axis defined by its direction cosines. The matrix for rotation is constructed in *v_rot_matx*. The elements of this are quite large but the overall work is minimised by calculating pairs of elements at a time due to the similarity of elements with their row and column indices interchanged.

In *vel_adj* the new direction of motion, which is the direction pointed to by the **kv** vector, is combined with the speed to produce a displacement of the view frame. What this amounts to is simply multiplying the components of kv by the speed and adding them to Ovx, Ovy and Ovz, the current value of the view frame origin in the world.

The test for visibility of objects follows the criteria explained above, where the object frame origin (Oox,Ooy,Ooz) is examined to see if it lies in the frustum defined as the field of view. To do this, the origin itself is first transformed into the view frame where it becomes (Vox,Voy,Voz).

One final routine, *scrn_adj*, is included to reset the centre of the screen at the origin of the world frame. This is not the same as simply moving the view frame in the world frame since it affects the appearance of perspective. Having the view frame centred on the screen is more natural to "flying around in space" experiences.

11.8.3 *bss_06.s*

This contains the few new variables introduced in this section: the base vectors and the rotations resulting from movement of the joystick.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                          wrld_vw.s                                      *
*            Joystick control of the view frame                           *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        SECTION TEXT
        opt     d+
        bra main
        include systm_02.s      screens and tables
        include systm_03.s      joystick
        include systm_04.s      set up screens, palette, joystick
        include core_06.s       new subroutines

main:
        bsr     set_up          set up screens etc
* transfer all the data
        bsr     transfer
        move.w  oncoords,vncoords
        move.w  vncoords,wncoords
* Initialise dynamical variables
        move.w  #0,Ovx          view frame
        move.w  #0,Ovy          starts off
        move.w  #-200,Ovz       200 behind world frame
* Set up view frame base vectors
* 1. iv
        lea     iv,a0           align
        move.w  #$4000,(a0)+    view
        clr.w   (a0)+           frame
        clr.w   (a0)            axes
* 2. jv
        lea     jv,a0           with
        clr.w   (a0)+           the
        move.w  #$4000,(a0)+    world
        clr.w   (a0)            frame
* 3. kv
        lea     kv,a0           axes
        clr.w   (a0)+
        clr.w   (a0)+
        move.w  #$4000,(a0)

        clr.w   speed           start at rest
        clr.w   screenflag      0=screen 1 draw, 1=screen 2 draw
        clr.w   viewflag
        bsr     clear1          clear the screens
        bsr     clear2
loop4:
* Switch the screens each time round
        tst.w   screenflag      screen 1 or screen2?
        beq     screen_1        draw on screen 1, display screen2
        bsr     draw2_disp1     draw on screen 2, display screen1
        bsr     clear2          but first wipe it clean
        clr.w   screenflag      and set the flag for next time
        bra     screen_2
screen_1:
        bsr     draw1_disp2     draw on 1, display 2
        bsr     clear1          but first wipe it clean
        move.w  #1,screenflag   and set the flag for next time
```

```
screen_2:
* Look for changes in the view frame angles.
        bsr     in_joy          read joystick and rotate the view frame
* See if the function keys have been pressed to change the speed.
        bsr     key_in
* Adjust to new velocity.
        bsr     vel_adj
* Recalculate the view frame base vectors and set up the world-view
* transform matrix.
        bsr     dircosines
* See if the object is within the visible angle of view.
        bsr     viewtest
        tst.b   viewflag        is it visible?
        beq     loop4           no, try again
* Construct the compound object from the same face at different positions.
        move.w  nparts,d7       how many parts in the object
        subq    #1,d7
        lea     inst_angles,a0  list of instance angles for each part
        lea     inst_disp,a1    ditto       displacements
* Do one face at a time
instance:
        move.w  d7,-(sp)        save the count
        move.w  (a0)+,otheta    next otheta
        move.w  (a0)+,ophi      next ophi
        move.w  (a0)+,ogamma    next ogamma
        move.w  (a1)+,Oox       next displacements
        move.w  (a1)+,Ooy
        move.w  (a1)+,Ooz
        movem.l a0/a1,-(sp)     save position in list
        bsr     otranw          object to world transform
        bsr     w_tran_v        world to view transform
        bsr     illuminate      if it's not hidden find the shade
        bsr     perspective     perspective
        bsr     scrn_adj        centre window
        bsr     polydraw        draw this face
        movem.l (sp)+,a0/a1     restore pointers
        move.w  (sp)+,d7        restore the parts count
        dbra    d7,instance     for all the parts of the object
        bra     loop4           draw the next frame

        SECTION DATA
        include data_03.s
        include data_05.s
        SECTION BSS
        include bss_06.s

        END
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                          core_06.s                                      *
*                 Subroutines for Chapter 11                              *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

        include core_05.s

dircosines:
*  Find the direction cosines for the transform from the world frame
* to the view frame. These are components of the view frame base
* vectors in the world frame. To avoid accumulating errors they
* are regenerated and normalised to a magnitude of 2^14.
        lea     iv,a0   here
        lea     jv,a1   they
        lea     kv,a2   are
* First kv is normalised
        move.w  (a2),d0
        move.w  2(a2),d1
        move.w  4(a2),d2
        bsr     nrm_vec         normalise it
        move.w  d0,(a2)         the
        move.w  d1,2(a2)        new
        move.w  d2,4(a2)        components
* Second vj is calculated from the cross product of vk
* and vi using the subroutine AxB: requires A pointer in a2
* B pointer in a0
        bsr     AxB
        move.w  d0,(a1)         regenerated
        move.w  d1,2(a1)        components
        move.w  d2,4(a1)
* Finally the cross product of kv and jv is used for iv
        lea     jv,a2
        lea     kv,a0
        bsr     AxB
        lea     iv,a1
        move.w  d0,(a1)         regenerated iv
        move.w  d1,2(a1)
        move.w  d2,4(a1)

* The components of the view frame base vectors in the world frame
* are the elements of the transform matrix required for the world-
* to-view transform.
        lea     w_vmatx,a0      pointer to the w-to-v matrix
        lea     iv,a1           here are
        lea     jv,a2           the view base
        lea     kv,a3           vectors
        move.w  (a1)+,(a0)+     and
        move.w  (a1)+,(a0)+     here
        move.w  (a1)+,(a0)+     are
        move.w  (a2)+,(a0)+     the
        move.w  (a2)+,(a0)+     matrix
        move.w  (a2)+,(a0)+     elements
        move.w  (a3)+,(a0)+     of the
        move.w  (a3)+,(a0)+     view
        move.w  (a3)+,(a0)+     transform
        rts
```

```
AxB:
* calculate the vector product AxB: pointer to A in a2, pointer to B
* in a0. Returns x-cmpt in d0, y-cmpt in d1, z-cmpt in d2.
* first component
        move.w  2(a2),d0        Ay
        muls    4(a0),d0        Bz*Ay
        move.w  4(a2),d1        Az
        muls    2(a0),d1        By*Az
        sub.l   d1,d0           Bz*Ay-By*Ax
* second component
        move.w  4(a2),d1        Az
        muls    (a0),d1         Bx*Az
        move.w  (a2),d2         Ax
        muls    4(a0),d2        Bz*Ax
        sub.l   d2,d1           Bx*Az-Bz*Ax
* third component
        move.w  (a2),d2         Ax
        muls    2(a0),d2        By*Ax
        move.w  2(a2),d3        Ay
        muls    (a0),d3         Bx*Ay
        sub.l   d3,d2           By*Ax-Bx*Ay
* reduce them to < word size by dividing by 2^14
        move    #14,d7
        lsr.l   d7,d0
        lsr.l   d7,d1
        lsr.l   d7,d2
* normalise them
        bsr     nrm_vec
        rts                     back to caller

* Do a rotation of the view frame about one of the view frame axes
* in the world frame. The direction cosines for the axis are
* the base vector components.
rot_vx:
* A rotation about the view frame x-axis, vx
        lea     iv,a0           the axis of rotation
        move.w  vxangle,d1      the angle to rotate
        bsr     v_rot_matx      construct the rotation matrix
* only jv and kv are affected
        lea     jv,a0           transform this first
        bsr     rot_view
        lea     kv,a0           transform this second
        bsr     rot_view
        rts
rot_vy:
* A rotation about the view frame y-axis, vy
        lea     jv,a0
        move.w  vyangle,d1
        bsr     v_rot_matx
* only iv and kv are affected
        lea     iv,a0
        bsr     rot_view
        lea     kv,a0
        bsr     rot_view
        rts
rot_vz:
* A rotation about the view frame z-axis, vz
        lea     kv,a0
        move.w  vzangle,d1
        bsr     v_rot_matx
```

```
* only iv and jv are affected
        lea     iv,a0
        bsr     rot_view
        lea     jv,a0
        bsr     rot_view
        rts

rot_view:
* Rotate a view frame base vector. The vector is pointed to by a0.
* Since it is a unit vector it is specified by three components
* which are the direction cosines (nx,ny,nz)
        moveq   #2,d6           3 rows in the transform matrix
        lea     vrot_matx,a3    init matrix pointer
        link    a6,#-6          3 words to store temporarily
rot_vw1 move.w  (a0),d0         nx component
        move.w  2(a0),d1        ny
        move.w  4(a0),d2        nz
        muls    (a3)+,d0        nx*Mi1
        muls    (a3)+,d1        ny*Mi2
        muls    (a3)+,d2        nz*Mi3
        add.l   d1,d0           add them
        add.l   d2,d0
        lsl.l   #2,d0           divide by 2^14
        swap    d0              the new component
        move.w  d0,-(a6)        save it
        dbra    d6,rot_vw1      repeat for three components
        move.w  (a6)+,4(a0)     off my stack into z
        move.w  (a6)+,2(a0)                     y
        move.w  (a6)+,(a0)                      x
        unlk    a6              release frame pointer
        rts

* Construct the rotation matrix for rotations about an arbitrary axis
* specified by a unit vector with components (direction cosines)
* (n1,n2,n3)
* Entry: pointer to direction cosines (n1,n2,n3), in a0,
* angle of rotation in d0.w
v_rot_matx:
        lea     vrot_matx,a6    pointer to the rotation matrix
        bsr     sincos          find the rotation sine and cosine
        move.w  d2,d6           sine delta
        move.w  d3,d7           cos delta

* elements M12 and M21
        move    #16384,d5
        move    d5,d0           1
        move.w  (a0),d1         n1
        muls    2(a0),d1        n1*n2
        lsl.l   #2,d1
        swap    d1
        sub.w   d7,d0           1-cosdelta
        move    d0,d4           save it
        muls    d1,d0
        lsl.l   #2,d0
        swap    d0              n1*n2(1-cosdelta)
        move    d0,d2
        move.w  4(a0),d1        n3
        muls    d6,d1           n3*sindelta
        lsl.l   #2,d1
```

```
        swap    d1
        sub.w   d1,d0           n1*n2(1-cosdelta)-n3*sindelta
        move.w  d0,2(a6)        M12
        add.w   d1,d2           n1*n2(1-cosdelta)+n3*sindelta
        move.w  d2,6(a6)        M21
* elements M13 and M31
        move    d4,d0           1-cosdelta
        muls    (a0),d0         n1*(1-cosdelta)
        lsl.l   #2,d0
        swap    d0
        muls    4(a0),d0        n1*n3(1-cosdelta)
        lsl.l   #2,d0
        swap    d0
        move    d0,d2
        move.w  2(a0),d1        n2
        muls    d6,d1           n2*sindelta
        lsl.l   #2,d1
        swap    d1
        add.w   d1,d0           n1*n3(1-cosdelta)+n2*sindelta
        move.w  d0,4(a6)        M13
        sub.w   d1,d2           n1*n3(1-cosdelta)-n2*sindelta
        move.w  d2,12(a6)       M31
* elements M23 and M32
        move    d4,d0           1-cosdelta
        muls    2(a0),d0        n2*(1-cosdelta)
        lsl.l   #2,d0
        swap    d0
        muls    4(a0),d0        n2*n3(1-cosdelta)
        lsl.l   #2,d0
        swap    d0
        move    d0,d2
        move.w  (a0),d1         n1
        muls    d6,d1           n1*sindelta
        lsl.l   #2,d1
        swap    d1
        sub.w   d1,d0           n2*n3(1-cosdelta)-n1*sindelta
        move.w  d0,10(a6)       M23
        add.w   d1,d2           n2*n3(1-cosdelta)+n1*sindelta
        move.w  d2,14(a6)       M32
* element M11
        move.w  (a0),d1         n1
        muls    d1,d1           n1*n1
        lsl.l   #2,d1
        swap    d1              /2^14
        move    d5,d2           1
        sub.w   d1,d2           1-n1*n1
        muls    d7,d2           (1-n1*n1)cosdelta
        lsl.l   #2,d2
        swap    d2
        add.w   d2,d1           n1*n1 +(1-n1*n1)cosdelta
        move.w  d1,(a6)         M11
*element M22
        move.w  2(a0),d1                n2
        muls    d1,d1           n2*n2
        lsl.l   #2,d1
        swap    d1              /2^14
        move    d5,d2           1
        sub.w   d1,d2           1-n2*n2
        muls    d7,d2           (1-n2*n2)cosdelta
```

```
        lsl.l   #2,d2
        swap    d2
        add.w   d2,d1           n2*n2 +(1-n2*n2)cosdelta
        move.w  d1,8(a6)        M22
*element M33
        move.w  4(a0),d1                n3
        muls    d1,d1           n3*n3
        lsl.l   #2,d1
        swap    d1              /2^14
        move    d5,d2
        sub.w   d1,d2           1-n3*n3
        muls    d7,d2           (1-n3*n3)cosdelta
        lsl.l   #2,d2
        swap    d2
        add.w   d2,d1           n3*n3 +(1-n3*n3)cosdelta
        move.w  d1,16(a6)       M33
        rts

* The the world coords are transformed to view coords
* Remember matrix elements are *2^14 and must be corrected at the end
w_tran_v:
        move.w  wncoords,d7     the number
        ext.l   d7              any to do ?
        beq     w_tranv3                if not quit
        subq.w  #1,d7           or this is the count

        lea     wcoordsx,a0     the
        lea     wcoordsy,a1     source
        lea     wcoordsz,a2     coords.
        lea     vcoordsx,a3     the
        lea     vcoordsy,a4     destination
        lea     vcoordsz,a5
        exg     a3,d3           save this address-short of regs.
        link    a6,#-6          3 words to store
w_tranv1:
        moveq.l #2,d6           3 rows in the matrix
        lea     w_vmatx,a3      init matx pointer
* calculate the next vx, vy and vz
w_tranv2:
        move.w  (a0),d0         wx
        move.w  (a1),d1         wy
        move.w  (a2),d2         wz

        sub.w   Ovx,d0          wx-Ovx
        sub.w   Ovy,d1          wy-Ovy
        sub.w   Ovz,d2          wz-Ovz

        muls    (a3)+,d0        wx*Mi1
        muls    (a3)+,d1        wy*Mi2
        muls    (a3)+,d2        wz*Mi3

        add.l   d1,d0
        add.l   d2,d0           wx*Mi1+wy*Mi2+wz*Mi3
        lsl.l   #2,d0
        swap    d0              /2^14
        move.w  d0,-(a6)        save it
        dbra    d6,w_tranv2     repeat for 3 elements
```

```
        move.w  (a6)+,(a5)+     off my stack becomes vz
        move.w  (a6)+,(a4)+     becomes vy
        exg     a3,d3           restore address vx, save matx pointr
        move.w  (a6)+,(a3)+     becomes vx
        exg     a3,d3           save address vx, restore matx pointr
        addq.l  #2,a0           point to next wx
        addq.l  #2,a1                         wy
        addq.l  #2,a2                         wz

        dbf     d7,w_tranv1     repeat for all ocoords
        unlk    a6              close frame
w_tranv3:
        rts                     and quit

in_joy:
* Read the joystick and update the variables accordingly
* The data packet containing the FIRE bit (7) and the position
* bits (0-2) is saved in the variable joy_data
        clr.w   joy_data
        move.w  #10,d6
more_joy:
        bsr     rd_joy          read joystick
        dbf     d6,more_joy     give it time to think
        move.w  joy_data,d0     here's the result
        move    d0,d1           save it
        andi.w  #$f0,d0         fire pressed  ?
        bne     jy_fire_press   yes
        andi.w  #$f,d1          what direction is the stick?
        bne     dir_joy
        rts                     nothing doing
dir_joy lea     table_jump,a0   base address
        lsl.w   #2,d1           offset into jump table
        move.l  0(a0,d1.w),a0   the jump address
        jmp     (a0)            go for it
table_jump:
        dc.l    0,jy_up,jy_down,0,jy_left,jy_up_left,jy_down_left
        dc.l    0,jy_right,jy_up_right,jy_down_right
jy_up   move.w  #350,vyangle    rotate up
        bsr     rot_vy          about vy axis
        rts
jy_down move.w  #10,vyangle     rotate down
        bsr     rot_vy          about vy axis
        rts
jy_left move.w  #10,vxangle     rotate left
        bsr     rot_vx          about vx axis
        rts
jy_right:
        move.w  #350,vxangle    rotate right
        bsr     rot_vx          about vx axis
        rts
jy_up_left      rts             do nothing for now
jy_down_left    rts
jy_up_right     rts
jy_down_right   rts
jy_fire_press:
        move.w  #1,fire
        rts
```

```
* Set the velocity components
vel_adj:
        lea     kv,a0
        moveq.l #14,d7          ready to divide by 2^14
        move.w  speed,d0
        lsl.w   #3,d0           scale it
        move    d0,d1
        move    d0,d2
        muls    (a0),d0         v*VZx
        lsr.l   d7,d0           /2^14
        add.w   d0,Ovx          xw speed component
        muls    2(a0),d1        v*VZy
        lsr.l   d7,d1
        add.w   d1,Ovy          zw speed component
        muls    4(a0),d2        v*VZz
        lsr.l   d7,d2
        add.w   d2,Ovz
        rts

viewtest:
* Test whether the primitive is visible. See whether its centre
* (Oox,Ooy,Ooz) lies within the angle of visibility.
* Oox, Ooy and Ooz are transformed to view coords then tested.
* Remember matrix elements are *2^14 and must be corrected at the end
        moveq.l #2,d6           3 rows in the matrix
        lea     w_vmatx,a3      init matx pointer
        link    a6,#-6          3 words to store temporarily
        move.w  Oox,d3          Oox     the
        addi.w  #50,d3
        move.w  Ooy,d4          Ooy     object centre
        addi.w  #50,d4
        move.w  Ooz,d5          Ooz
        addi.w  #50,d5
        sub.w   Ovx,d3          Oox-Ovx relative to the view frame
        sub.w   Ovy,d4          Ooy-Ovy
        sub.w   Ovz,d5          Ooz-Ovz
tranOv  move    d3,d0           restore
        move    d4,d1
        move    d5,d2
        muls    (a3)+,d0        *Mi1
        muls    (a3)+,d1        *Mi2
        muls    (a3)+,d2        *Mi3
        add.l   d1,d0
        add.l   d2,d0           *Mi1+*Mi2+*Mi3
        lsl.l   #2,d0
        swap    d0              /2^14
        move.w  d0,-(a6)        save it
        dbra    d6,tranOv       repeat for 3 elements

        move.w  (a6)+,d3        off my stack becomes Voz
        move.w  (a6)+,d2        becomes Voy
        move.w  (a6)+,d1        becomes Vox
        move.w  d3,Voz
        move.w  d2,Voy
        move.w  d1,Vox
        unlk    a6              close frame
* Clip Ovz. For visibility must have 100<Voz<2000
        cmpi.w  #100,d3         test (Voz-100)
        bmi     invis           fail
        cmpi.w  #2000,d3        test(Voz-2000)
```

```
        bpl     invis           fail
* Is it within the allowed angle of view?
        addi.w  #100,d3         Voz+100
        add.w   d3,d3           *2
        add.w   d3,d3           *4
        add.w   d3,d3           *8
* first test horizontal position
        tst.w   d2              is Voy +ve or -ve?
        bpl     pos_y           it's +ve
        neg.w   d2              it's -ve so make it +ve for test
pos_y   cmp.w   d2,d3           Voy is +ve, test(8*(Voz+100)-Voy)
        bmi     invis           Voy too big
* second test vertical position
        tst.w   d1              Vox
        bpl     pos_x           it's +ve
        neg.w   d1              make it +ve
pos_x   cmp.w   d1,d3           test(8(Voz+100)-Vox)
        bmi     invis           too high
* It is visible
        st      viewflag        set the flag all 1's
        rts
* It's invisible, don't draw it
invis   sf      viewflag        set the flag all 0's
        rts

scrn_adj:
* adjust screen coords so that view frame (0,0) is at the centre
        move.w  vncoords,d7     the number
        beq     adj_end         quit if none
        subq.w  #1,d7           count
        lea     scoordsy,a0     y coords pointer
adj_loop:
        subi.w  #100,(a0)+      adjust next ys
        dbra    d7,adj_loop     for all points
adj_end rts
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                bss_06.s                                 *
*                Variables for Chapter 11                                 *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        include bss_05.s
* variables for rotating the view frame
iv      ds.w    3       view frame base vector components in world frame
jv      ds.w    3
kv      ds.w    3
vxangle ds.w    1       rotation angles about these axes
vyangle ds.w    1
vzangle ds.w    1
vrot_matx       ds.w    9       rotation matx. about an arbitrary axis
* visibility
viewflag        ds.w    1
Vox     ds.w    1       object centre in view frame
Voy     ds.w    1
Voz     ds.w    1
```

12

A World Scene

In this chapter a world containing many objects is constructed.

The transition from a single graphics primitive to a scene containing several brings a host of new problems. For example, in the complex scene of many objects, spatial relationships must be preserved; objects in the foreground must not be obscured by those in the distance. Some form of depth sorting is required that orders objects for drawing on the basis of their distance from the observer.

Just as important is a sound strategy for ignoring all objects outside the immediate environment of the observer. In a world consisting of hundreds of objects spread out over a landscape, it would be pointlessly time consuming to attempt to draw them all. As in real life, the observer need only be concerned with those that are close by and affect current decisions. We examine these aspects of the multi-object world in turn.

12.1 A Database

Associated with each object in the complex world will be a list of its attributes (type, position, colour, rotation angles, etc.), and the set of lists of all the objects is a database. It contains all information needed to draw the view seen by the observer. Exactly how this database is laid out in memory is very important in determining the speed with which it can be accessed for graphics.

To explain this point further, consider the choices available in ordering the objects in the database. Objects could be entered in the database in order of increasing x (world) coordinate or increasing y coordinate or increasing z coordinate, or indeed at random with no spatial order whatsoever. Objects could be listed according to their type, colour or any one of their attributes. Of all the possibilities there will be

those that provide fast access to those objects which are going to be drawn, i.e. those in the immediate vicinity of the observer. It is clear that some kind of ordering in position is needed to achieve this.

12.1.1 A Map

The position of an object in the world is specified by its three coordinates in the form (xw,yw,zw). It is clear that ordering the database in any one single coordinate (xw or yw or zw) alone will not provide an immediate picture of where each object is in relation to its neighbours.

What is needed is a database where the objects are arranged in 3D order. This is difficult to visualise until it is realised that what is being described is nothing more than a map. The similarity to an ordinary route map is fairly exact for the world we will construct which consists of objects sitting on a surface, just like the surface of the Earth. The advantage of a map of this kind, (which is a 2D array) is that all the objects that lie in a particular region are immediately obvious in their spatial relations.

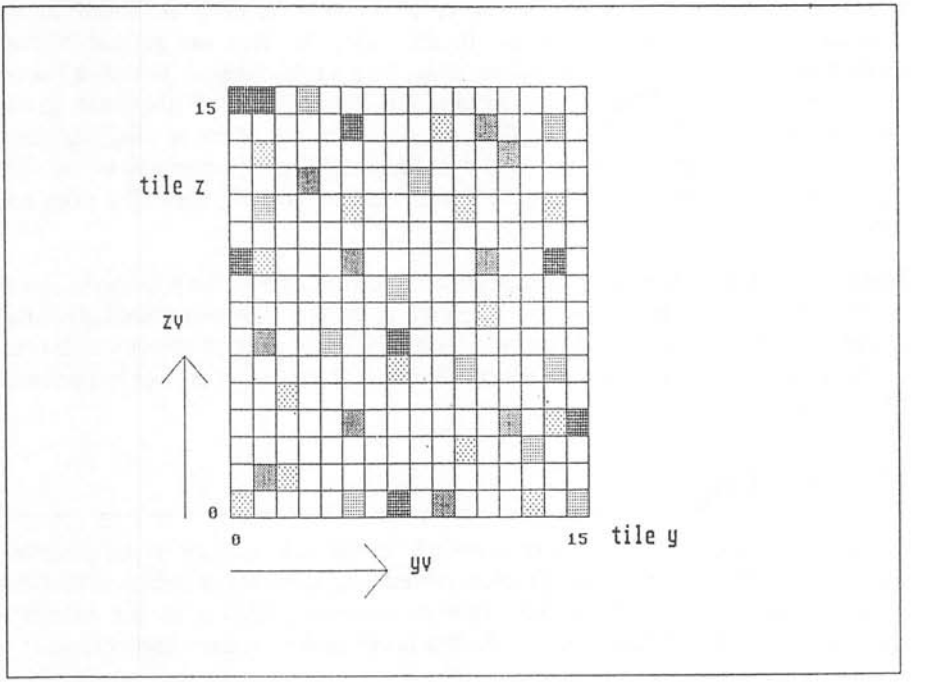

Figure 12.1 Layout of world 'tiles'

What is actually done is shown in Figure 12.1. The world space is divided into a 16*16 array of "tiles" (just like on the bathroom wall) each one of which has the dimensions 256*256. Each tile is a unit of space to be considered for display. It can contain a collection of objects; in the example program it contains just one, for simplicity. Of course this is not a very extensive world, but there is nothing in the method which limits it to these dimensions; it could be a big as you like and the individual tiles as small as you like. But, "wrap" occurs so that when the observer strays off any edge he reappears on the opposite side; in this way the world is effectively "infinite", like a sphere. For our purposes a 16*16 tile world is sufficient to illustrate the method. Each tile defines a region of space which, for the purposes of display, is a single entity. To construct the view seen by the observer, all that has to be done is to find her/his position on the tile grid, select the nearest-neighbour tiles, find which ones are in front of the observer and draw the objects placed on them.

How can this 2D array be laid out in the 1D contiguous RAM? There is nothing new here. The screen itself is a 2D world which is represented in memory as a 1D database. The pixel is analogous to a tile and the four bits which specify its colour are analogous to the data list specifying the attributes of the object on the tile. An arrangement of information in this way, where each element is linked to its adjacent ones is called a linked list. In this case, the links are permanent and implied by the physical position in the array. The world database is thus a list of 256 bytes, each one holding the attributes of one tile in the 16*16 tile world. In the example program it is held in the file data_08.s. The list starts at map_base and every 16th byte starts a new tile in the z direction. The tile position in the list, mod16, represents the 16 y values. In this model the world is flat and x does not vary.

There is very little information needed for the attributes, since the position in space is automatically included by the tile's position in the list. The first nibble gives the colour of the background (1-15) and the second gives the type of object which is to sit on the tile. At present only six are possible (listed in *data_06.s*), but in principle there is no limit.

12.2 Sorting

As mentioned above, once the visible objects in the near vicinity to the observer have been identified there is the problem of ordering them for drawing so that the more distant ones are drawn first. This is commonly known as the painter's algorithm, since in painting a picture the last brush stroke overlays earlier ones.

There are many well known algorithms for sorting data in order. Most of the more exotic varieties have been developed to handle large databases with a large number of entries (records). In our case it is necessary to sort a small number (<16) of

records in depth order. Sorting at this level is efficiently done by one of the simplest sorting methods, called a bubble sort. Note that at this stage we are referring to the attributes and other accumulated data about the objects to be drawn as records. A record is a set of data of different types where each data type is confined to specific parts or "fields". This is how data for visible objects is carried around in the example programs. A record is constructed containing all the relevant data to draw in the tile and during depth sorting the records are actually sorted like a deck of cards. That way, although the depth field is the basis for sorting, it carries with it other information for drawing, reducing the retrieval of additional data at a later stage to a minimum. Of course, to avoid slowing things down too much it's important to keep the record short. In the example program a record consists of 2 long words divided into 7 fields.

12.2.1 A Bubble Sort

Let's illustrate the bubble sort by direct example from the program. In this we have a short list of records for the visible objects to be displayed. The field on which the sort is based is the second word in the record. It is the distance of the object from the origin of the view frame in the positive z direction, i.e. the direction in which the viewer is looking. The other fields are unimportant for the sorting. Figure 12.2

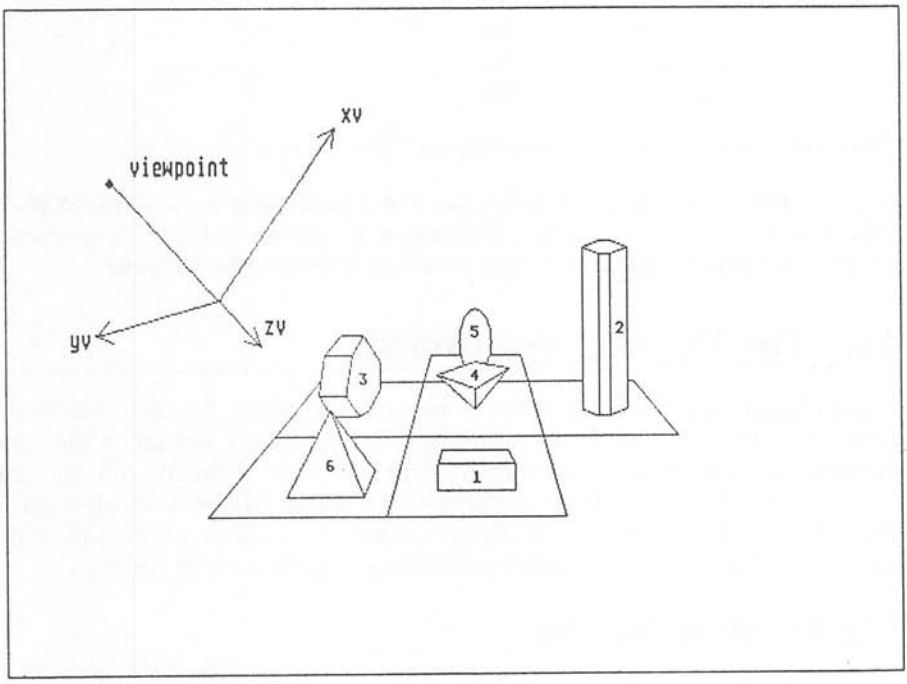

Figure 12.2 Depth ordering of objects

shows a possible arrangement of simple objects in front of the view frame. The number on each object is its type, which is the content of the second field on its record. A suitable order in which they should be drawn so that objects in the rear lie behind those in the forefront is: 2,1,4,5,6,3. But this is unlikely to be the order in which the tiles have been retrieved from the database. Let us suppose that they have been withdrawn in the order 6,1,3,4,2,5. The sorting now begins.

The procedure in a bubble sort is to go through the list comparing each entry with its successor and making a switch if necessary. In the present case we will order the list with the objects to be drawn first at the top of the list, i.e. the list will be in the order: distant objects - near objects. In the first sweep, first the first pair 6 and 1 are examined, found to be in the wrong order and exchanged. At the same time, to record that the list was found to be out of order, a flag is set. This leaves 1 as the first entry and 6 as the second. Then the next pair 6 and 3 are examined. The order here is O.K so no switch is made. This is continued through the entire list. Each time a switch is made the flag is set (of course it can only be set once so the following swaps do nothing to the flag). The following lines show the progression of the first sort:

6,1,3,4,2,5	start
1,6,3,4,2,5	1st pair tested
1,6,3,4,2,5	2nd
1,6,4,3,2,5	3rd
1,6,4,2,3,5	4th
1,6,4,2,5,3	5th.

Notice how, like bubbles, the distant objects "float" to the top.

At the end of the list the flag is tested to see if a switch was made. If so the entire list is tested again. This is repeated until a pass is made in which the flag was not set, in which case the list in order and the sort is deemed to be complete.

12.3 The Viewing Transform

In this chapter we include two different ways of constructing the view seen by the observer. The first uses control matrices and is a simpler version of the view transform used in the previous chapter. The second is altogether different and much simpler; it uses the Euler angles met in Chapter 10 and is widely used in elementary flight simulators. It is slightly limited as a consequence of the way the angles are defined. We discuss the application of control matrices first.

12.3.1 Control Matrices

Let us suppose that we have reached the stage where all the transforms have been done to present a scene from the viewpoint of an observer. The vertices of all

visible objects will then be given in the frame of reference of the observer, i.e. the view frame. If, as a consequence, for example, of a movement of the joystick the observer moves his head to the left, all that is required to show the new view is to rotate the vertices to the right. Rotation of the observer about any axis in his reference frame can be implemented by rotating view frame vertices coordinates in the opposite direction. Such a transform is called a coordinate transform since it calculates the view seen from a different coordinate system, i.e. the rotated coordinate system of the observer.

So it seems that all that is required to show the view of the observer, as he flies through the world, is to multiply the view frame coordinates by the sequence of rotation matrices representing his accumulated motion to date. It won't work! First a record of the total sequence of rotations would have to be kept and then, for each frame, they would have to be multiplied out in order. Not exactly an efficient algorithm for fast graphics. After a while the picture would stop altogether as hundreds of matrix multiplications were done for each frame. What is the solution?

The solution to this problem is very similar to the method used in the previous chapter where the view frame base vectors were rotated and then used to construct the view transform. In this case the procedure is done backwards. At any instant, as a result of calculations done to display the previous frame, we know the view transform matrix. This is the starting point for the next frame. The sequence of events at the end of the calculations will be to: 1) do the view transform to convert vertices to the view frame, 2) do the rotations about view frame axes we have been talking about, 3) finally, do the perspective transform and everything else that follows. Here is now the solution to the problem. Instead of regarding the view transform, (V), and the view frame rotations, (C), as separate transforms, to be done to the vertices, (PW), in the world frame in sequence to produce first the view frame vertices (PV) and then the rotated vertices (PV').

$$(C)(V)(PW) = (C)(PV) = (PV'),$$

we concatenate (multiply out) (C) and (V) separately beforehand

to produce a rotated view transform, (V')

$$(C)(V)(PW) = (V')(PW) = (PV').$$

In this scheme each rotation of the observer is brought about by pre-multiplying the view transform by a "control" matrix appropriate to the rotation. The control matrices for the separate rotations about the view frame xv, yv and zv axes are:

$$(Cx) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{pmatrix}$$

$$(Cy) = \begin{pmatrix} \cos\phi & 0 & -\sin\phi \\ 0 & 1 & 0 \\ \sin\phi & 0 & \cos\phi \end{pmatrix}$$

$$(Cz) = \begin{pmatrix} \cos\gamma & \sin\gamma & 0 \\ -\sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Notice that these are exactly the same as the geometric transforms of Chapter 7 except that the sine terms have the opposite sign. This is because

$$\sin(-\theta) = -\sin(\theta)$$

and shows that the coordinate transforms are the same as geometric transforms with negative angles, i.e. they correspond to backward rotations. This is saying mathematically what we know to be true: rotating the observer's head to the left achieves the same end result as rotating the scene to the right. (See Appendix 7).

The physical motions corresponding to the rotations are shown in Figure 12.3. They are: yaw (rotation about the x axis), pitch (rotation about the y axis) and roll (rotation about the z axis).

To speed things up the control matrices can be precalculated. If it is accepted that rotations always occur in 1 degree increments then the elements of the matrices will be sine(1) and cos(1) (multiplied by 16384 as usual). This is indeed what is done in the example program file *dat_07.s* where angle increments are taken to be 5, although here rotations only occur about the xv and yv axes.

There still remains the need to ensure that errors do not accumulate. So, remembering that the rows of the view transform can be visualised as the view frame base vectors, we regenerate the view matrix rows by vector products as was done in Chapter 11.

The details of all these stages are shown in the example program, wrld_scn.

12.3.2. Euler Angles

We have already discussed these in section 10.1.1. Euler angles are a way of specifying the orientation of one reference frame with respect to another using only three angles but with some restriction as to how the angles are defined. Most important is that they specify rotations about different axes in a fixed order. There are many combinations possible. The sequence defined below is the one beloved

of aeronautical engineers and is called the 321 sequence because it describes rotations about the x, y, and z axes in order. These correlate with motions of the joystick and so describe yaw (bearing), pitch and roll but note that yaw here, being an initial rotation about the world frame axis, wx, is different from that described in section 12.3.1. The physical rotations of the observer are shown in Figure 12.3.

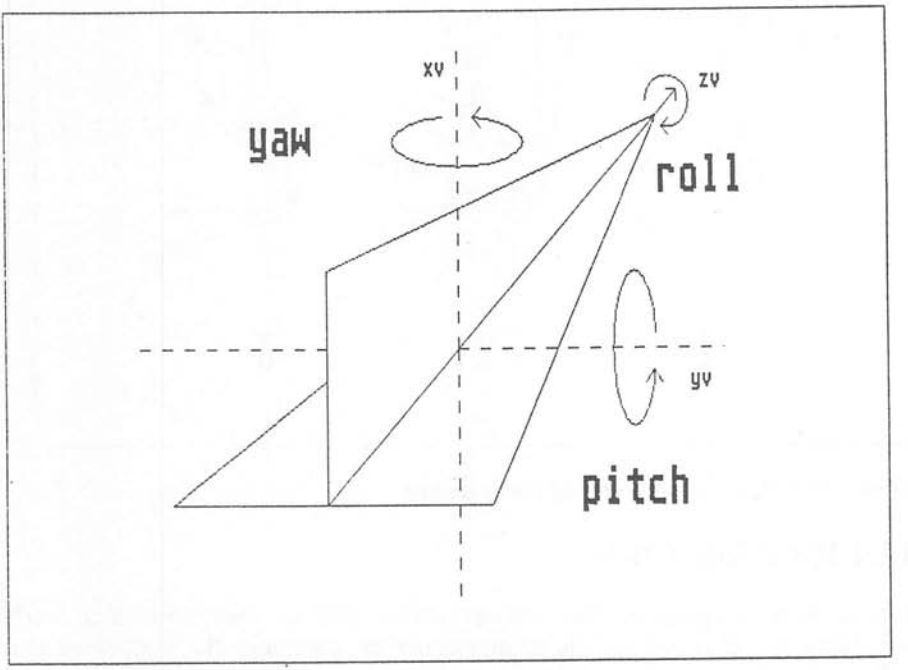

Figure 12.3 Aeronautical terms for viewframe rotations

Here is the sequence of rotations (displacements have already been subtracted off) which carry the world reference frame into the observer's view frame. It is illustrated in Figure 12.4. Both frames are coincident to begin with and rotations are about view frame axes, wherever they are at the time:

1. rotate by θ about the x axis – the same for both frames (yaw)
2. rotate by ϕ about the y axis (pitch)
3. rotate by γ about the z axis (roll)

The end product is the orientation of the view frame.

Looking back to section 10.1.1. it will be seen that this is precisely the sequence of rotations done there and so the results, in particular the final matrix product, can be used directly. The results are illustrated in the example program *eulr_scn*.

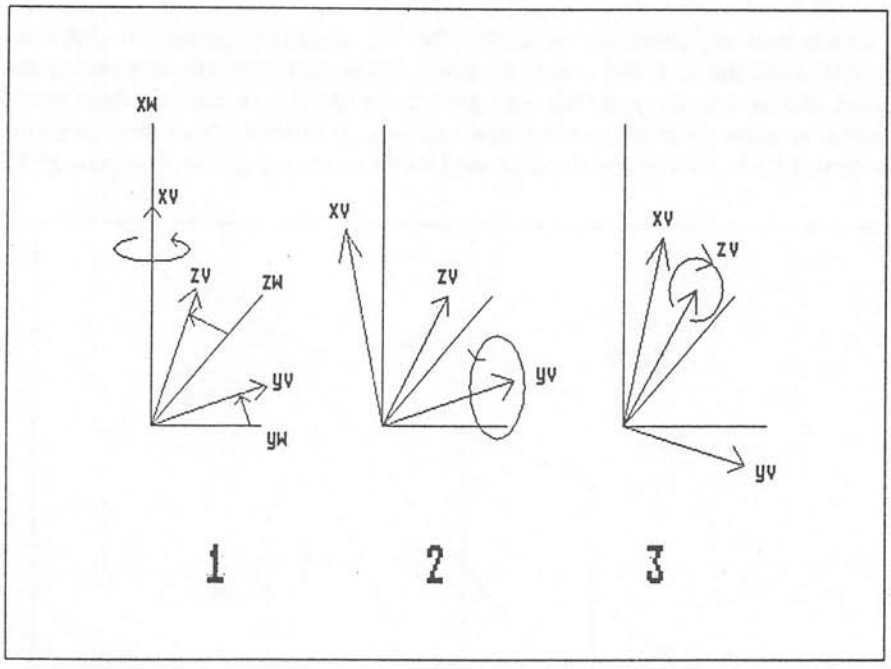

Figure 12.4 Rotation sequence of Euler angles

12.4 Running Times

The example program in this chapter allows you to roam around a world containing 256 different graphic entities under the control of the joystick as in a rudimentary flight simulator. There is no limitation here; a larger world database could be constructed with no additional time penalty. A world of this limited size has been used because it is sufficient to illustrate the procedures involved without involving excessively long listings.

Because of the serial way the book has introduced the different stages of getting a moving picture on the screen, and the manner in which programs have been included together to make an overall program of increasing power, there has been an inevitable compromise in speed. The final program in this last chapter could be rationalised and simplified to become substantially faster.

12.5 Example Program

12.5.1 *wrld_scn.s* and *eulr_scn.s*

There are two main control programs here. They both allow free flight through a landscape of moving objects but differ in the type of viewing transform used. In

one of them, wrld_scn.s, motion is controlled through the joystick and keyboard by means of rotations about the instantaneous axes of the observer's coordinate frame. In the other, eulr_scn.s the joystick increments or decrements the Euler angles and to vary the orientation of the observer's reference frame. The detailed controls are

wrld_scn: up, down, left, right = joystick
roll left = f1, roll right = f2

eulr_scn: up, down, left, right = joystick.

In both cases the other function keys are:

reverse=f3, slow forward=f4, fast forward=f5, stop=f6, abort=f7.

12.5.2 *data_06.s*

This is the data file of the graphics primitives, which are simple 3D structures. They appear littered about the landscape according to the database in *data_08.s* where the primitive associated with each tile is specified in the low nibble of the attribute byte. There are 6 types (0-5) vectored from a jump table at the address primitive. There is no limit to the variety or number; to include a new one simply add one more label to the jump vectors and fill in the details at the end of the list. The primary jump vectors at primitive point to a list of secondary vectors, which are the tables of data for each particular type

For a particular type data is given in a series of lists:

- the secondary pointers,
- the intrinsic colours (0 or 1 for 8 shades of 2 colours),
- the number of faces on each polyhedral object,
- the list of edge numbers on each face,
- the list of vertex connections on all faces in order,
- the three sets of x, y and z coordinates of the vertices,
- the total number of vertices and
- the type of rotation which the object is undergoing.

The type of rotational motion which each type displays is specified in the lowest nibble of the high word of the variable θn (where n is the type number) and the low word is used by the program to hold the current angle but appears as 0 in the list. The type of rotation is given by the bit which is set in the nibble:

bit 0 - rotation about x axis of object frame

bit 1 - ditto y

bit 2 - ditto z

so that any combination of simultaneous rotations can be included.

12.5.3 *data_07.s*

Here are the four control matrices for positive and negative rotations about the view frame x and y axes laid out in row order.

12.5.4 *data_08.s*

Here are the 256 bytes which make up the 16*16 tile world unit. In the program, wrap-around occurs so that motion beyond the extreme left boundary returns the viewer to the right boundary. In this sense, like a sphere, the world is "infinite". In each byte the high nibble gives the actual colour of the background ((0-7), no illumination) and the high nibble gives the object type (0-15) sitting on the tile. Only 6 types are used in the program. The reader can easily invent new ones.

12.5.5 *core_07.s*

The first subroutine in the core, *patch_ext* first takes the observer's current position and normalises it to lie within the world map. This is where the wrap-around occurs. Following this the location in tile coordinates (Ty,Tz) is calculated by dividing by the y and z positions by 256. Remember there are 16*16 tiles spread out over the y-z plane. This is the vertical projection of the observer's position onto the plane. Then the attributes of the 16 tiles centred about this position are retrieved from the database and, for each tile, stored as the first byte of the first word in the 4-word record which accompanies each one. The offset of each tile from the observer's position is saved in the second byte of the first record word. This collection of potentially visible tiles is called a patch.

Following this a visibility test is done on every tile in the patch. The test here does not consider a frustum of visibility, but only whether the centre of the tile lies in front of the observer. The central parameter calculated for each tile during this test is its distance (zv) in front of the observer. This is also saved as the second word in the record for depth sorting later. Less than half the tiles pass the visibility test. The visibility sort, next, simply uses a bubble sort to place the records in order of depth, that is in order of decreasing distance from the observer. The tiles with records at the top of the list will be drawn first since they are farthest away.

The subroutine which follows, *drw_it*, sets up the data to draw each tile and its resident object in the ordered list of visible tiles, and calls all the earlier

subroutines to draw the complete picture. There is a lot going on at this stage. The background on each tile is just a cross of a particular colour so that all the tiles together define a grid on which the objects sit. Since the background is the same for every tile, it is entered directly from the program rather than being stored in a data file. Also since it has a fixed colour without varying illumination, there is no need to call the time-consuming illumination calculations.

The data lists for each object are pulled in from the data file and before it is drawn its new angle in the world frame is determined for whatever mode of rotation is active.

12.5.6 *bss_07.s*

New variables

12.5.7 *systm_05.s*

Just a few routines to set up the system. In particular the view point is moved back a bit to -300 on the zv axis to reduce the perspective distortion and eliminate the possibility of parts of objects falling behind the observer, which would not cause the system to crash but would produce a display of spectacular garbage as the basic drawing routines attempt to cope with drawing backwards.

Also a bit of a cheat. The ST is being stretched to its limit with this program and it helps to speed things up by reducing the size of the window (clip frame) on the screen so that the picture is smaller (ever wondered why games show a tiny screen surrounded by a lot of static ornamentation looking like a console?).

12.5.8 core_08.s

This is the core file for the Euler angle transform.

12.6 Epilogue

How far have we got? What's next?

For a start the overall program can be speeded up considerably by rationalising the anomalies caused by the serial way in which programs have been introduced in this book.

There also remains the inclusion of the third party (you, the world scene and the alien). So far the graphic entities have been static in the sense that their evolution has been determined by their attributes. To give entities life requires that their actions evolve independent of the deterministic structure of the program. But there is really only one truly random element in this scenario - you, the observer. Hence

to create life within the computer it is necessary to make the entities respond to your actions. This is of course what happens in all games. Aliens head for the target. To invent a third party is no more complicated than has already been done in reading the movements of the joystick to follow the motion of the observer. In the case of the third party there are no joystick movements, but rather, the response to world conditions.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                        wrld_scn.s                                   *
*                  A multi-object scene                               *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

* A world scene consisting of various types of graphics primitives
* in motion. The viewer is free to "fly" to any location. At any
* position a patch consisting of 4*4 "tiles" is visible.
* Joystick controls Yaw and pitch. F1 and F2 control roll
* Don't hold down keys as keyboard buffer is not cleared.

        SECTION TEXT
        opt     d+
        bra main
        include systm_05.s
        include core_07.s

main:
* Initialize the system.
        bsr     init_vars       initialise view transform
        bsr     flg_init        initialize flags

loop:
* Read input and make adjustments.
        bsr     swap_scn        swap the screens
        bsr     dircosines      regenerate view matrix
        bsr     joy_read        see which direction to move
        bsr     in_key          update the speed
        bsr     adj_vel         adjust the velocity

* Draw the scene
        bsr     scne_drw        everything to complete the picture

* Draw the next frame
        bra     loop

        SECTION DATA
        include data_06.s
        include data_07.s
        include data_08.s
        SECTION BSS
        include bss_07.s

        END
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                        core_07.s                                      *
*                  Subroutines for Chapter 12                           *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

        include core_06.s

scne_drw:
* Draw the scene of several primitives
        bsr     patch_ext       select the local scene
        bsr     sight_tst       select only visible ones
        bsr     vis_srt         sort them in depth order
        bsr     drw_it          draw them in depth order
        rts

patch_ext:
* Extract the tile patch. Put the 16 tiles in a list at patch_lst
        move.w  oposx,d0        observer x position
        move.w  oposy,d1        y
        move.w  oposz,d2        z
* Find position in world. Keep to range 4096
        andi.w  #$fff,d0        range x
        andi.w  #$fff,d1        range y
        andi.w  #$fff,d2        range z
        move.w  d0,oposx        restore x
        move.w  d1,oposy        y
        move.w  d2,oposz        z
        move.w  d1,d3
        move.w  d2,d4
* find coords of patch centre = local world origin
        lsr.w   #8,d1           /256
        move.w  d1,Ty           y coord. in 16*16 layout
        lsr.w   #8,d2           /256
        move.w  d2,Tz           z coord
* coords of view frame referenced to this origin
        lsl.w   #8,d1           Ty*256
        lsl.w   #8,d2           Tz*256
        sub.w   d1,d3           oposy-Ty*256 = Ovy
        move.w  d3,Ovy
        sub.w   d2,d4           oposz-Tz*256 = Ovz
        move.w  d4,Ovz
        move.w  oposx,Ovx       Ovx (the height is universal)

* Fetch the attributes of the 16 surrounding tiles from the map
* and calculate their world coords., storing the data in a record
* with the format:
* word 1:       high byte = graphics attributes
*               low byte  = clear
* word 2:       Voz tile centre z in view frame coords
* word 3:       tile y in local world coords
* word 4:       ditto z
* Ty and Tz are the patch centre coords. = local world origin
        move.w  Ty,d0           Ty
        move.w  Tz,d1           Tz
* A 4*4 patch of tiles centred on the Ty,Tz are retrieved
        move.w  #-2,d5          z offset of start tile
        lea     map_base,a0     pointer to map of 16*16=256 tiles
        lea     patch_lst,a1    the local list of 4*4
        move.w  #3,d7           4 z values
```

```
tile_lp1:
        move.w  #-2,d4          reset start yoffset
        move.w  #3,d6           4 y values
        move.w  d1,d3           origin Tz
        add.w   d5,d3           + offset = next z
        andi.w  #$f,d3          stay in range 0-15
        lsl.w   #4,d3           *16
tile_lp2:
        move    d0,d2           origin Ty
        add.w   d4,d2           + offset = next y
        andi.w  #$f,d2          stay in range 0-15
        add.w   d3,d2           16*z+y = tile address in map
        move.b  0(a0,d2.w),d2   fetch attribute in low byte
        swap    d2              of high word
        clr.w   d2              0 for low word
        lsl.l   #8,d2           everything into high word
        move.l  d2,(a1)+        store the 1st half of the record
* Calculate the tile local coords.: Ooy and Ooz
* Local coords are (offset*256)
        movem.l d4/d5,-(sp)     stack offsets
        lsl     #8,d4           yoffset*256
        swap    d4              in high word
        lsl     #8,d5           zoffset*256
        move.w  d5,d4           in low word
        move.l  d4,(a1)+        store second half of record
        movem.l (sp)+,d4/d5     restore offsets
        addq    #1,d4           next y offset
        dbra    d6,tile_lp2     for all the tiles in this row
        addi.w  #1,d5           next z offset
        dbra    d7,tile_lp1     for all rows
        rts
sight_tst:
* Discard all tiles which are out of sight
        lea     patch_lst,a0    pointer to source list
        lea     vis_lst,a1      pointer to list of visible tiles
        lea     vis_cnt,a2      pointer to count of visible tiles
        clr.w   (a2)            set count to zero
        move.w  #15,d7          16 tiles in a patch
        clr.w   Oox             all tiles are on ground
sight_tst1:
        move.w  4(a0),d0
        addi.w  #128,d0
        move.w  d0,Ooy          tile
        move.w  6(a0),d0
        addi.w  #128,d0
        move.w  d0,Ooz          centres
        movem.l d7/a0-a2,-(sp)
        bsr     testview        is this tile within the field of view?
        movem.l (sp)+,d7/a0-a2
        tst.b   viewflag        visible?
        beq     nxt_tile        no
        addq.w  #1,(a2)         yes, increment visible count
        move.w  Voz,2(a0)       save the depth for sorting
        move.l  (a0),(a1)+      transfer 1st half to visible list
        move.l  4(a0),(a1)+     transfer 2nd half of record
```

```
nxt_tile:
        addq    #8,a0           point to next record
        dbra    d7,sight_tst1   for all tiles
        rts

testview:
* Is the tile within the field of view?
* Test whether the primitive is visible.
* Tile centre (Oox,Ooy,Ooz) is transformed to view coords then tested.
* (remember matrix elements are *2^14 and must be corrected at the end)
        moveq.l #2,d6           3 rows in the matrix
        lea     w_vmatx,a3      init matx pointer
        link    a6,#-6          3 words to store temporarily
        move.w  Oox,d3          Oox     the
        move.w  Ooy,d4          Ooy     object centre
        move.w  Ooz,d5          Ooz
        sub.w   Ovx,d3          Oox-Ovx relative to the view frame
        sub.w   Ovy,d4          Ooy-Ovy
        sub.w   Ovz,d5          Ooz-Ovz
tranv0  move    d3,d0           restore
        move    d4,d1
        move    d5,d2
        muls    (a3)+,d0        *Mi1
        muls    (a3)+,d1        *Mi2
        muls    (a3)+,d2        *Mi3
        add.l   d1,d0
        add.l   d2,d0           *Mi1+*Mi2+*Mi3
        lsl.l   #2,d0
        swap    d0              /2^14
        move.w  d0,-(a6)        save it
        dbra    d6,tranv0       repeat for 3 elements

        move.w  (a6)+,d3        off my stack becomes Voz
        move.w  (a6)+,d2        becomes Voy (the centre in view frame)
        move.w  (a6)+,d1        becomes Vox
        move.w  d3,Voz
        move.w  d2,Voy
        move.w  d1,Vox
        unlk    a6              close frame
* Clip Ovz. To be visible, must have 50<Voz<2000
* This visibility test looks only at depth
        cmp.w   #50,d3          test(Voz-50)
        bmi     notvis          fail
        cmp.w   #2000,d3        test(Voz-2000)
        bpl     notvis          fail
        st      viewflag        it's visible, set the flag all 1's
        rts
* It's invisible, don't draw it
notvis  sf      viewflag        set the flag all 0's
        rts
```

```
vis_srt:
* Order the visible tiles in order of decreasing Voz.
* Voz is the distance of the tile centre from the view frame
* origin. Largest Voz's should be drawn first.
        move.w  vis_cnt,d7      number to do
        beq     srt_quit        are any visible?
        subq    #1,d7           this number-1
        beq     srt_quit        but no need to sort only 1
        subq.w  #1,d7           1 sort per pair
* bubble sort the records
vis_srt1:
        lea     vis_lst+2,a0    pointer to the first record Voz
        movea.l a0,a1
        addq.l  #8,a1           pointer to second record Voz
        move    d7,d6           reset count
        clr.w   srt_flg clear the flag
vis_srt2:
        cmpm.w  (a0)+,(a1)+     test(Voz2-Voz1) and advance pointer
        ble     no_swap         1st is farther
        move.l  -4(a0),d0       fetch 1st record
        move.l  (a0),d1
        move.l  -4(a1),-4(a0)   make
        move.l  (a1),(a0)       second first and
        move.l  d0,-4(a1)       first
        move.l  d1,(a1)         second
        st      srt_flg         set the flag
no_swap:
        addq.l  #6,a0           point to next record Voz
        addq.l  #6,a1           and the one following
        dbra    d6,vis_srt2     for all records.
        tst.w   srt_flg         Were any records swapped?
        beq     srt_quit        no
        bra     vis_srt1        yes, run through again
srt_quit:
        rts                     sort is finished

drw_it:
* Draw the visible tiles
        move.w  vis_cnt,d7
        beq     drw_it_out
        subq.w  #1,d7
        lea     vis_lst,a0      pointer to list
drw_it1:
        movem.l d7/a0,-(sp)
        bsr     set_prim        draw the next primitive
        movem.l (sp)+,d7/a0
        addq.l  #8,a0           next record
        dbra    d7,drw_it1
drw_it_out:
        rts
```

```
set_prim:
* set up next primitive for drawing; enters with pointer to record in a0
* 1. First do the background
        move.l  a0,-(sp)        save pointer
        bsr     ldup_bkg        load background data as program data
        bsr     otranw          object-to-world
        bsr     w_tran_v        world-to-view
* It's always visible at constant illumination; pass colour directly
        movea.l (sp)+,a0        restore pointer
        move.w  (a0),d0         first word of record
        move.l  a0,-(sp)        save pointer
        lsr.w   #8,d0           top byte
        lsr.w   #4,d0           top nibble is colour
        move.w  d0,col_lst      the final
        move.w  d0,col_lst+2    colours
        bsr     perspective
        bsr     scrn_adj        centre it
        bsr     polydraw        draw it
* 2. Second draw the object
        movea.l (sp)+,a6        restore pointer
        bsr     ldup_obj        load object data as program data
        bsr     otranw          object-to-world
        bsr     w_tran_v        world-to-view
        bsr     illuminate      all
        bsr     perspective     the
        bsr     scrn_adj        rest
        bsr     polydraw
        rts

ldup_bkg:
* Load background data as program data. The background is a grid.
        move.w  #2,npoly        2 polygons (intersecting rectangles)
        move.l  #$40004,snedges 4 edges in each
        lea     sedglst,a2      edge list 0,1,2,3,0,4,5,6,7,4
        move.l  #1,(a2)+        edges 0,1
        move.l  #$20003,(a2)+   edges 2,3
        move.l  #$4,(a2)+       edges 0,4
        move.l  #$50006,(a2)+   edges 5,6
        move.l  #$70004,(a2)+   edges 7,4
* the background vertices define a cross
* all x coords are zero
        lea     ocoordsx,a2     vertex coords x =
        move.l  #0,(a2)+                0,0
        move.l  #0,(a2)+                0,0
        move.l  #0,(a2)+                0,0
        move.l  #0,(a2)                 0,0
        lea     ocoordsy,a2             y =
        move.l  #$ff800080,(a2)+        -128,128
        move.l  #$80ff80,(a2)+          128,-128
        move.l  #$fffcfffc,(a2)+        -4,-4
        move.l  #$40004,(a2)            4,4
        lea     ocoordsz,a2             z =
        move.l  #$40004,(a2)+           4,4
        move.l  #$fffcfffc,(a2)+        -4,-4
        move.l  #$ff800080,(a2)+        -128,128
        move.l  #$80ff80,(a2)           128,-128
        move.w  #8,oncoords     the
        move.w  #8,vncoords     counts
        move.w  #8,wncoords     are all the same
```

```
* the tile centre in the world frame is Oox=0 and the
* contents of the third and fourth words of the record
        move.w  #0,Oox
        move.w  4(a0),Ooy       third word
        addi.w  #128,Ooy
        move.w  6(a0),Ooz       fourth word
        addi.w  #128,Ooz
        clr.w   otheta          no
        clr.w   ophi            orientation
        clr.w   ogamma
        rts                     all done
* Load background data as program data
        move.w  #1,npoly        only one polygon
        move.w  #4,snedges      4 edges
        lea     sedglst,a2      edge list 0,1,2,3,0
        move.l  #1,(a2)+        edge 0,1
        move.l  #$20003,(a2)+   edge 2,3
        clr.w   (a2)+           edge 0
* the background vertices are the corners of the tile
        lea     ocoordsx,a2     vertex coords x =
        move.w  #0,(a2)+                0,0
        move.w  #0,(a2)+                0,0
        move.w  #0,(a2)+
        move.w  #0,(a2)+
        lea     ocoordsy,a2     y =
        clr.l   (a2)+           0,0
        move.l  #$ff00ff,(a2)   255,255
        lea     ocoordsz,a2     z =
        move.l  #$ff,(a2)+      0,255
        move.l  #$ff0000,(a2)   255,0
        move.w  #4,oncoords     the
        move.w  #4,vncoords     counts
        move.w  #4,wncoords     are all the same
* the tile centre in the world frame is Oox=0 and the
* contents of the third and fourth words of the record
        move.w  #0,Oox
        move.w  4(a0),Ooy       third word
        move.w  6(a0),Ooz       fourth word
        clr.w   otheta          no
        clr.w   ophi            orientation
        clr.w   ogamma
        rts                     all done

ldup_obj:
* Load object data as program data
* first find out what type it is; pointer to record in a6
        move.w  (a6),d0         top word
        lsr.w   #8,d0           top byte
        andi.w  #$f,d0          bottom nibble is type (call it n)
        lsl.w   #2,d0           *4 for offset
        lea     primitive,a5    pointer to vector table
        movea.l 0(a5,d0.w),a5   pointer to type n lists
        movea.l 4(a5),a2        pointer to npolyn
        move.w  (a2),d7         here it is
        move.w  d7,npoly
        subq.w  #1,d7           the count
        move    d7,d0           save it
        movea.l 8(a5),a0        pointer to nedgn (list edge numbers)
        movea.l a0,a4           save it
        lea     snedges,a1      destination
```

```
        move.l  (a5),a2         pointer to intrinsic colours
        lea     srf_col,a3      destination
obj_lp1 move.w  (a0)+,(a1)+     transfer edge numbers
        move.w  (a2)+,(a3)+     transfer intrinsic colours
        dbra    d0,obj_lp1
* calculate the total number of edges
        move.w  d7,d0           restore count
        clr     d1
        clr     d2
obj_lp2 add.w   (a4)+,d2        number of edges
        addq    #1,d2           and with last repeated
        dbra    d0,obj_lp2
*move the edge list
        subq    #1,d2           this is the counter
        movea.l 12(a5),a0       edglstn, the source
        lea     sedglst,a1      destination
obj_lp3 move.w  (a0)+,(a1)+     pass it
        dbra    d2,obj_lp3
* and the coordinates list
        movea.l 28(a5),a0       pointer to no. vertices
        move.w  (a0),d1         no. vertices
        move.w  d1,oncoords     same
        move.w  d1,vncoords     for
        move.w  d1,wncoords     all frames
        subq    #1,d1           the counter
        movea.l 16(a5),a0       pointer object x
        lea     ocoordsx,a1
        movea.l 20(a5),a2       object y
        lea     ocoordsy,a3
        movea.l 24(a5),a4       object z
        movea.l a5,a6
        lea     ocoordsz,a5
obj_lp4 move.w  (a0)+,(a1)+
        move.w  (a2)+,(a3)+
        move.w  (a4)+,(a5)+
        dbra    d1,obj_lp4
* increment the rotation angle
        bsr     next_rot
        addi.w  #128,Ooy
        addi.w  #128,Ooz
        rts

* Increment the rotation of the object
next_rot:
        movea.l 32(a6),a0       pointer to angle and flag
        move.l  (a0),d0         top word is flag, bottom is angle
        move.l  d0,d1
        andi.l  #$ffff,d0       the angle
        addi.w  #2,d0           increment it
        cmp     #360,d0
        blt     obj_lp5
        subi    #360,d0
obj_lp5 move.w  d0,2(a0)        next angle
* see what angles to rotate
        swap    d1
        andi.w  #$f,d1          the flag is in the low nibble
```

```
* flags are set: bit 0 = x rot, bit 1 = y rot, bit 2 = z rot
        lsl.w   #2,d1           offset
        lea     rot_vec,a0      pointer to jump table
        move.l  0(a0,d1.w),a0   the jump vector
        jmp     (a0)            here goes
rot_vec:
        dc.l    no_rot,rotx,roty,rotxy,rotz,rotxz,rotyz,rotxyz
no_rot  rts                     no rotation
rotx    move.w  d0,otheta       rotate about x axis
        rts
roty    move.w  d0,ophi                 y
        rts
rotxy   move.w  d0,otheta               x and y
        move.w  d0,ophi
        rts
rotz    move.w  d0,ogamma               z
        rts
rotxz   move.w  d0,otheta               x and z
        move.w  d0,ogamma
        rts
rotyz   move.w  d0,ophi                 y and z
        move.w  d0,ogamma
        rts
rotxyz  move.w  d0,otheta               x, y and z
        move.w  d0,ophi
        move.w  d0,ogamma
        rts

joy_read:
* Rotate the view point about an axis
* Read the joystick and update the variables accordingly
* The data packet containing the FIRE bit (7) and the position
* bits (0-2) is saved in the variable joy_data
        clr.w   joy_data
        move.w  #10,d6
joy_more:
        bsr     rd_joy          read joystick
        dbf     d6,joy_more     give it time to think
        move.w  joy_data,d0     here's the result
        move    d0,d1           save it
        andi.w  #$f0,d0         fire pressed  ?
        bne     jy_press_fire   yes
        andi.w  #$f,d1          what direction is the stick?
        bne     joy_dr
        rts                     nothing doing
joy_dr  lea     jump_joy,a0     base address
        lsl.w   #2,d1           offset into jump table
        move.l  0(a0,d1.w),a0   the jump address
        jmp     (a0)            go for it
jump_joy:
        dc.l    0,up_jy,down_jy,0,left_jy,up_left_jy,down_left_jy
        dc.l    0,right_jy,up_right_jy,down_right_jy
up_jy   bsr     rot_down        rotate view frame down about vy axis
        rts
down_jy bsr     rot_up          rotate up about vy axis
        rts
left_jy bsr     rot_left        rotate left about vx axis
        rts
```

```
right_jy:
        bsr     rot_right       rotate right about vx axis
        rts
up_left_jy      rts             do nothing for now
down_left_jy    rts
up_right_jy     rts
down_right_jy   rts
jy_press_fire:
        move.w  #1,fire
        rts

in_key:
* Read the keyboard to set view frame speed
        bsr     scan_keys       was a key pressed?
        cmp.w   #-1,d0
        beq     ky_read yes
        rts                     no
ky_read:
        bsr     read_key        which key?
        tst.w   d0              f keys have $0 in the low word
        beq     ky_rpt          only interested if f keys
        rts                     something else
ky_rpt  swap    d0              the code
        subi.w  #$3b,d0         f1 is 3b : set it to zero for offset
        andi.w  #7,d0           first 7 f keys
        lea     ky_jump,a0      jump table
        lsl.w   #2,d0           key code is offset
        movea.l 0(a0,d0.w),a0   to the routine address
        jmp     (a0)            go for it
ky_jump:
* The jump table for f keys
        dc.l    f1f,f2f,f3f,f4f,f5f,f6f,f7f
f1f     bsr     roll_left       roll to the left
        rts
f2f     bsr     roll_right      roll to the right
        rts
f3f     move.w  #-2,speed       reverse speed 2
        rts
f4f     move.w  #2,speed        forward speed 2
        rts
f5f     move.w  #3,speed        forward speed 3
        rts
f6f     move.w  #0,speed        stop
        rts
f7f     clr.w   -(sp)
        trap    #1              quit altogether- return to caller

rot_down:
* Rotate down about the yv axis. Multiply the "down" control matrix
* by the view transform matrix.
        lea     rot_y_neg,a0    pointer to the control matrix
        bsr     ctrl_view       multiply and set base vectors
        rts

rot_up:
* Rotate up about the yv axis. Multiply the "up " control
* matrix by the view transform matrix.
        lea     rot_y_pos,a0
        bsr     ctrl_view
        rts
```

```
rot_left:
* Rotate left about the xv axis . Multiply the "left" control
* matrix by the view transform matrix.
        lea     rot_x_pos,a0
        bsr     ctrl_view
        rts

rot_right:
* Rotate right about the xv axis. Multiply the "right" control
* matrix by the view transform matrix.
        lea     rot_x_neg,a0
        bsr     ctrl_view
        rts

roll_left:
* Rotate left about the zv axis. Multiply the "roll-left"
* control matrix by the view transform matrix.
        lea     rot_z_neg,a0
        bsr     ctrl_view
        rts

roll_right:
* Rotate right about the zv axis. Multiply the "roll-right"
* control matrix by the view transform matrix.
        lea     rot_z_pos,a0
        bsr     ctrl_view
        rts

ctrl_view:
* Multiply the control matrix pointed to by a0 by the view matrix
* to calculate the new elements of the view base vectors.
*1. base vector iv
        lea     w_vmatx,a1      pointer to view matrix
        lea     iv,a2           pointer view frame base vector iv
        move.w  #2,d6           3 elements to iv
        movea.l a1,a3           set view pointer
iv_loop move.w  (a3),d1         next view element
        move.w  6(a3),d2        "
        move.w  12(a3),d3       "
        muls    (a0),d1
        muls    2(a0),d2
        muls    4(a0),d3
        add.l   d2,d1
        add.l   d3,d1
        lsl.l   #2,d1           /
        swap    d1              2^14
        move.w  d1,(a2)+        next element in base vector
        addq.l  #2,a3           next column in view matrix
        dbra    d6,iv_loop      for all elements in this base vector

*2. No need to do the base vector jv; it is calculated from the other two.
```

```
*3. base vector kv
        lea     kv,a2           pointer view frame base vector kv
        move.w  #2,d6           3 elements to kv
        movea.l a1,a3           reset view pointer
kv_loop move.w  (a3),d1         next view element
        move.w  6(a3),d2        "
        move.w  12(a3),d3       "
        muls    12(a0),d1
        muls    14(a0),d2
        muls    16(a0),d3
        add.l   d2,d1
        add.l   d3,d1
        lsl.l   #2,d1           /
        swap    d1              2^14
        move.w  d1,(a2)+        next element in base vector
        addq.l  #2,a3           next column in view matrix
        dbra    d6,kv_loop      for all elements in this base vector
        rts                     all done

* Set the velocity components
adj_vel:
        lea     kv,a0
        move.w  #14,d7          ready to divide by 2^14
        move.w  speed,d0
        lsl.w   #4,d0           scale it
        move    d0,d1
        move    d0,d2
        muls    (a0),d0         v*VZx
        lsr.l   d7,d0           /2^14
        add.w   d0,oposx        xw speed component
        bpl     adj1
        clr.w   oposx           oposx must be >0
adj1    muls    2(a0),d1        v*VZy
        lsr.l   d7,d1
        add.w   d1,oposy        yw speed component
        muls    4(a0),d2        v*VZz
        lsr.l   d7,d2
        add.w   d2,oposz        zw speed component
        rts
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                       *
*                          bss_07.s                                     *
*                  variables for Chapter 12                             *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        include bss_06.s
* Observer position in world (mod4096)
oposx   ds.w    1
oposy   ds.w    1
oposz   ds.w    1

* Tile offset in 16*16 patch
Ty      ds.w    1
Tz      ds.w    1

* Tile lists
patch_lst       ds.l    32      records (8 byte) of 16 tiles in patch
vis_lst         ds.l    32      records of visible tiles

* List vars
vis_cnt ds.w    1               number of visible tiles
srt_flg ds.w    1       set during sorting in depth order
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                          data_06.s                                    *
*                  Data file for Chapter 12                             *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        include data_03.s
        include data_05.s

* the vector table of graphics primitives
primitive:
        dc.l    prim0,prim1,prim2,prim3,prim4,prim5

* the vector table for primitive #0. A simple block.
prim0   dc.l    colrs0,npoly0,nedg0,edglst0,prm0x,prm0y,prm0z,npts0
        dc.l    theta0
colrs0  dc.w    1,1,1,1,1               8 shades of 1 colour
npoly0  dc.w    5                       block
nedg0   dc.w    4,4,4,4,4
edglst0 dc.w    0,1,2,3,0,3,2,4,5,3,5,4,6,7,5,7,6,1,0,7,1,6,4,2,1
prm0x   dc.w    0,50,50,0,70,0,70,0
prm0y   dc.w    -6,-6,6,6,6,6,-6,-6
prm0z   dc.w    -6,-6,-6,-6,6,6,6,6
npts0   dc.w    8
theta0  dc.l    $10000

* the vector table for primitive #1. An inverted pyramid.
prim1   dc.l    colrs1,npoly1,nedg1,edglst1,prm1x,prm1y,prm1z,npts1
        dc.l    theta1
colrs1  dc.w    1,1,1,1,0               sides and top differ in colour
npoly1  dc.w    5
nedg1   dc.w    3,3,3,3,4
edglst1 dc.w    0,1,2,0,0,2,3,0,0,3,4,0,0,4,1,0,1,4,3,2,1
prm1x   dc.w    0,75,75,75,75
prm1y   dc.w    0,-32,32,32,-32
prm1z   dc.w    0,-32,-32,32,32
npts1   dc.w    5
theta1  dc.l    $10000

* the vector table for primitive #2. A nugget.
prim2   dc.l    colrs2,npoly2,nedg2,edglst2,prm2x,prm2y,prm2z,npts2
        dc.l    theta2
colrs2  dc.w    1,1,0,1,0,0,1,0,1,1,0,1,0,1
npoly2  dc.w    14
nedg2   dc.w    4,4,4,4,4,4,4,4,4,4,4,4,4,4
edglst2 dc.w    1,6,4,2,1,0,1,2,3,0,3,2,4,5,3,4,6,7,5,4,6,1,0,7,6
        dc.w    8,0,3,11,8,3,5,10,11,3,5,7,9,10,5,7,0,8,9,7,8,11,13,12,8
        dc.w    11,10,14,13,11,10,9,15,14,10,9,8,12,15,9,12,13,14,15,12
prm2x   dc.w    40,60,60,40,60,40,60,40,20,20,20,20,0,0,0,0
prm2y   dc.w    -30,-10,10,30,10,30,-10,-30,-30,-30,30,30,-10,10,10,-10
prm2z   dc.w    -30,-10,-10,-30,10,30,10,30,-30,30,30,-30,-10,-10,10,10
npts2   dc.w    16
theta2  dc.l    $70000
```

```
* the vector table for primitive #3. A Tee.
prim3   dc.l    colrs3,npoly3,nedg3,edglst3,prm3x,prm3y,prm3z,npts3
        dc.l    theta3
colrs3  dc.w    1,1,1,1,1,1,1,1,1,1
npoly3  dc.w    10
nedg3   dc.w    4,4,4,4,4,4,4,4,4,4
edglst3 dc.w    0,1,2,3,0,3,2,4,7,3,4,5,6,7,4,5,1,0,6,5
        dc.w    8,11,14,15,8,13,14,11,10,13,12,13,10,9,12,8,15,12,9,8
        dc.w    12,15,14,13,12,10,11,8,9,10
prm3x   dc.w    0,45,45,0,45,45,0,0,70,45,45,70,45,45,70,70
prm3y   dc.w    -10,-10,10,10,10,-10,-10,10,128,128,128,128,-128,-128
        dc.w    -128,-128
prm3z   dc.w    -10,-10,-10,-10,10,10,10,10,10,10,-10,-10,10,-10,-10,10
npts3   dc.w    16
theta3  dc.l    $10000

* the vector table for primitive #4. A roller.
prim4   dc.l    colrs4,npoly4,nedg4,edglst4,prm4x,prm4y,prm4z,npts4
        dc.l    theta4
colrs4  dc.w    1,0,1,0,1,0,1,1
npoly4  dc.w    8
nedg4   dc.w    4,4,4,4,4,4,6,6
edglst4 dc.w    1,2,8,7,1,0,1,7,6,0,5,0,6,11,5,4,5,11,10,4,3,4,10,9,3
        dc.w    2,3,9,8,2,4,3,2,1,0,5,4,6,7,8,9,10,11,6
prm4x   dc.w    0,40,40,0,-40,-40,0,40,40,0,-40,-40
prm4y   dc.w    -32,-32,-32,-32,-32,-32,32,32,32,32,32,32
prm4z   dc.w    -45,-20,20,45,20,-20,-45,-20,20,45,20,-20
npts4   dc.w    12
theta4  dc.l    $20000

* the vector table for primitive #5. Another roller.
prim5   dc.l    colrs5,npoly5,nedg5,edglst5,prm5x,prm5y,prm5z,npts5
        dc.l    theta5
colrs5  dc.w    1,0,1,0,1,0,1,1
npoly5  dc.w    8
nedg5   dc.w    4,4,4,4,4,4,6,6
edglst5 dc.w    1,2,8,7,1,0,1,7,6,0,5,0,6,11,5,4,5,11,10,4,3,4,10,9,3
        dc.w    2,3,9,8,2,4,3,2,1,0,5,4,6,7,8,9,10,11,6
prm5x   dc.w    0,40,40,0,-40,-40,0,40,40,0,-40,-40
prm5y   dc.w    -8,-8,-8,-8,-8,-8,8,8,8,8,8,8
prm5z   dc.w    -45,-20,20,45,20,-20,-45,-20,20,45,20,-20
npts5   dc.w    12
theta5  dc.l    $40000
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                        data_07                                        *
*                control matrices for rotation                          *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

* +ve rotation about the view frame x axis (left) by.5 degrees
rot_x_pos:
        dc.w    16384,0,0,0,16322,1428,0,-1428,16322

* -ve rotation about the xv axis (right)
rot_x_neg:
        dc.w    16384,0,0,0,16322,-1428,0,1428,16322

* +ve rotation about the yv axis (up)
rot_y_pos:
        dc.w    16322,0,-1428,0,16384,0,1428,0,16322

* -ve rotation about the yv axis (down)
rot_y_neg:
        dc.w    16322,0,1428,0,16384,0,-1428,0,16322

* +ve rotation about the zv axis (roll-right)
rot_z_pos:
        dc.w    16322,1428,0,-1428,16322,0,0,0,16384

* -ve rotation about the zv axis (roll-left)
rot_z_neg:
        dc.w    16322,-1428,0,1428,16322,0,0,0,16384
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                          data_08.s                                    *
*                  The world layout for Chapter 12                      *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

* The map of the world.
* Each byte gives the attributes of a size 256*256 "tile" in a
* 16*16 tile world. The attribute is broken down:
* high nibble = background colour (1-7)
* low nibble = primitive type (0-5)
map_base:
        dc.b    $62,$62,$62,$50,$41,$35,$35,$35
        dc.b    $35,$35,$35,$43,$45,$54,$54,$64
        dc.b    $62,$62,$62,$55,$42,$33,$35,$35
        dc.b    $35,$35,$32,$44,$45,$54,$54,$64
        dc.b    $52,$52,$52,$52,$44,$35,$34,$35
        dc.b    $35,$30,$35,$41,$44,$54,$54,$64
        dc.b    $45,$41,$42,$42,$42,$35,$22,$23
        dc.b    $23,$20,$25,$25,$44,$44,$40,$65
        dc.b    $33,$35,$30,$32,$32,$22,$25,$25
        dc.b    $25,$23,$24,$24,$35,$32,$35,$31
        dc.b    $35,$32,$35,$35,$32,$22,$11,$11
        dc.b    $10,$10,$24,$24,$33,$35,$32,$34
        dc.b    $20,$25,$25,$25,$20,$21,$13,$13
        dc.b    $13,$13,$20,$25,$25,$25,$20,$25
        dc.b    $24,$25,$25,$25,$21,$21,$13,$13
        dc.b    $13,$13,$20,$20,$25,$25,$20,$25
        dc.b    $20,$25,$25,$25,$22,$22,$13,$13
        dc.b    $13,$13,$14,$24,$25,$25,$22,$23
        dc.b    $25,$23,$25,$25,$23,$22,$13,$13
        dc.b    $13,$13,$14,$23,$25,$25,$25,$25
        dc.b    $31,$35,$30,$35,$31,$21,$22,$22
        dc.b    $20,$20,$20,$20,$35,$35,$34,$33
        dc.b    $45,$40,$40,$40,$41,$41,$22,$22
        dc.b    $22,$25,$30,$40,$40,$42,$45,$41
        dc.b    $40,$40,$41,$41,$44,$45,$30,$35
        dc.b    $35,$35,$32,$45,$40,$50,$55,$55
        dc.b    $61,$61,$61,$51,$53,$45,$35,$32
        dc.b    $35,$35,$31,$45,$40,$50,$60,$60
        dc.b    $61,$61,$61,$52,$55,$44,$33,$35
        dc.b    $33,$35,$30,$45,$40,$50,$60,$60
        dc.b    $61,$61,$61,$55,$51,$45,$30,$35
        dc.b    $32,$35,$35,$41,$45,$50,$60,$60
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                          systm_05.s                                   *
*                  routines for Chapter 12                              *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        include systm_02.s
        include systm_03.s
        include systm_04.s
init_vars:
* set up the screens
        bsr     set_up
* set the view point
        move.w  #100,oposx
        clr.w   oposy
        clr.w   oposz
* and the clip frame
        move.w  #50,xmin
        move.w  #270,xmax
        move.w  #30,ymin
        move.w  #170,ymax
* Set up view frame base vectors
* 1. iv
        lea     iv,a0           align
        move.w  #16384,(a0)+    view
        move.w  #0,(a0)+                frame
        move.w  #0,(a0)         axes
* 2. jv
        lea     jv,a0           with
        clr.w   (a0)+           the
        move.w  #16384,(a0)+    world
        clr.w   (a0)            frame
* 3. kv
        lea     kv,a0           axes
        move.w  #0,(a0)+
        clr.w   (a0)+
        move.w  #16384,(a0)
flg_init:
* Initialize flags and other variables
        clr.w   speed           start off at rest
        clr.w   screenflag      0=screen 1 draw, 1=screen 2 draw
        clr.w   viewflag
* Move the view point to -300 on the view frame z axis
        lea     persmatx,a0
        move.w  #300,d0
        move.w  d0,(a0)
        move.w  d0,10(a0)
        move.w  d0,30(a0)
        rts
swap_scn:
        tst.w   screenflag      screen 1 or screen2?
        beq     screen_1        draw on screen 1, display screen2
        bsr     draw2_disp1     draw on screen 2, display screen1
        bsr     clear2          but first wipe it clean
        clr.w   screenflag      and set the flag for next time
        bra     screen_2
screen_1:
        bsr     draw1_disp2     draw on 1, display 2
        bsr     clear1          but first wipe it clean
        move.w  #1,screenflag   and set the flag for next time
screen_2:
        rts
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                        eulr_scn.s                                     *
*                   A multi-object scene                                *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

* A world scene consisting of various types of graphics primitives
* in motion. The viewer is free to "fly" to any location with
* flight simulator type control from the joystick. At any
* position a patch consisting of 4*4 "tiles" is visible.

        SECTION TEXT
        opt     d+
        bra main
        include systm_05.s
        include core_08.s

main:
* Initialize the system.
        bsr     init_vars       initialise view transform
        bsr     flg_init        initialize flags

loop:
* Read input and make adjustments.
        bsr     swap_scn        swap the screens
        bsr     joy_look        see which direction to move
        bsr     angle_update    change the euler angles
        bsr     wtranv_1        construct the view transform
        bsr     vtran_move      move it to the base vectors
        bsr     in_key          update the speed
        bsr     adj_vel         adjust the velocity

* Draw the scene
        bsr     scne_drw        everything to complete the picture

* Draw the next frame
        bra     loop

        SECTION DATA
        include data_06.s
        include data_07.s
        include data_08.s
        SECTION BSS
        include bss_07.s

        END
```

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                        core_08.s                                      *
*                Subroutines for eulr_scn, Chapter 12                   *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

        include core_07.s       previous subroutines

joy_look:
* Change the euler angles etheta and ephi (vtheta and vphi from
* Chapter 10 are same thing)
* Read the joystick and update the variables accordingly
* The data packet containing the FIRE bit (7) and the position
* bits (0-2) is saved in the variable joy_data
        clr.w   joy_data
        move.w  #10,d6
ejoy_more:
        bsr     rd_joy          read joystick
        dbf     d6,ejoy_more    give it time to think
        move.w  joy_data,d0     here's the result
        move    d0,d1           save it
        andi.w  #$f0,d0         fire pressed  ?
        bne     ejy_press_fire  yes
        andi.w  #$f,d1          what direction is the stick?
        bne     ejoy_dr
        rts                     nothing doing
ejoy_dr lea     ejump_joy,a0    base address
        lsl.w   #2,d1           offset into jump table
        move.l  0(a0,d1.w),a0   the jump address
        jmp     (a0)            go for it
ejump_joy:
        dc.l    0,eup_jy,edown_jy,0,eleft_jy,eup_left_jy,edown_left_jy
        dc.l    0,eright_jy,eup_right_jy,edown_right_jy
eup_jy  bsr     erot_down       rotate view frame down about vy axis
        rts
edown_jy        bsr     erot_up         rotate up about vy axis
        rts
eleft_jy        bsr     erot_left       rotate left about wx axis
        rts
eright_jy:
        bsr     erot_right      rotate right about wx axis
        rts
eup_left_jy     rts             do nothing for now
edown_left_jy   rts
eup_right_jy    rts
edown_right_jy  rts
ejy_press_fire:
        move.w  #1,fire
        rts

erot_down:
* Rotate down about the yv axis. Decrement ephi (same as vphi)
        move.w  #-5,vphi_inc
        rts

erot_up:
* Rotate up about the yv axis. Increment ephi (same as vphi)
        move.w  #5,vphi_inc
        rts
```

```
erot_left:
* Rotate left about the xw axis . Increment etheta
        move.w  #5,vtheta_inc
        rts

erot_right:
* Rotate right about the xw axis. Decrement etheta
        move.w  #-5,vtheta_inc
        rts

vtran_move:
* move the view transform matrix to the base vectors
* (really just a change of label)
        lea     iv,a0
        lea     jv,a1
        lea     kv,a2
        lea     w_vmatx,a3
        move.w  (a3)+,(a0)+     all
        move.w  (a3)+,(a0)+     iv
        move.w  (a3)+,(a0)
        move.w  (a3)+,(a1)+     all
        move.w  (a3)+,(a1)+     jv
        move.w  (a3)+,(a1)
        move.w  (a3)+,(a2)+     all
        move.w  (a3)+,(a2)+     kv
        move.w  (a3),(a2)
        rts
```

Appendix 1

68000 Instruction Set

Entire books have been written concerning the 68000 instruction set. There is insufficient space here to do more than outline the essentials. A succinct but thorough discussion is given in the Motorola 16-Bit User's Manual.

The central feature of assembly language programming is that there are no abstract algebraic variables as in regular mathematics or high level languages such as BASIC. It is not possible to make statements such as

```
LET x=y+z
```

though it is possible to effect equivalent manipulations of data.

In assembly language, names such as x, y or z are labels representing addresses in RAM. At these addresses can be found binary numbers which are the current values of the parameters associated with the labels. There is a similarity to algebraic variables but at every stage it is the binary number itself which is manipulated either in memory or in the processor registers. The addressing modes of the 68000 are designed to deal with all the ways data needs to be addressed or directed through the system during the execution of the various instructions.

The 68000 instruction set is extensive and powerful. It has two important aspects: the instructions themselves and their addressing modes, which form the basic framework for data acquisition and manipulation.

A1.1 Registers

The 68000 processor has eight 32-bit data registers (D0-D7) dedicated to data, seven 32-bit address registers which can be used for data and addresses (A0-A6), two 32-bit stack pointers (both called A7 but used separately, one for the system and one for the user) set to point to last-in, first-out temporary storage areas of

RAM (stacks), one 32-bit program counter to keep count of program progress and one 16-bit status register of flags to record results of operations. The 32-bit registers can be used to handle the five basic data types: bits, bcd digits, bytes (8 bits), words (16 bits) and long words (32 bits).

A1.2 Addressing Modes

Each instruction is concerned with the manipulation of data of some kind somewhere in the microcomputer system: in the processor, in memory or from external hardware. The addressing modes are designed for the many ways data is accessed. There are six basic types: Register Direct, Register Indirect, Absolute, Immediate, Program Counter Relative and Implied, which encompass the 14 modes listed below. For each instruction, the data (which can be an address) which is about to be manipulated, is located somewhere in the system. The addressing modes give the ways this location is to be found. In its most general form this to-be-determined address is called an effective address *(ea)*.

Addressing Modes

Immediate Data Addressing

Immediate	the data is the next word
Quick Immediate	the data is included with the instruction

Implied

ea = SR, SP or PC

Register Direct

Address Register Direct ea = An
(data contained in named address register)
Data Register Direct ea = Dn
(data contained in named data register)

Absolute Data Addressing.

Absolute Short ea = (next word)
(data is at address given at next word following instruction)
Absolute Long ea = (next 2 words)

Register Indirect Addressing

Register Indirect ea = (An)
(data is at address given in named address register)
Postincrement Register Indirect ea = (An)+
(as (An), then increment register)
Predecrement Register Indirect ea = -(An)
(as (An) but predecrement register)

Register Indirect with Offset (as (An) plus a word length addition)	ea = d16(An)
Indexed Register Indirect with Offset (As (An) plus a byte length addition together with the contents of an address or data register acting as an index)	ea = d8(An,Xn)

An important version of register indirect addressing is *PC relative*, where the program counter is used instead of An in d16(An) and d8(An,Xn). This allows reference to memory locations relative to the current program counter and is used to generate position independent code. It is not used in this book since the assembler generates relocatable code which achieves the same end.

Instruction Set

In general instructions have associated with them a source operand and a destination operand. What these actually mean depends specifically on the instruction, for example in a MOVE instruction they do exactly what they imply - supply the source and destination effective addresses. In an ADD instruction they give the addresses of the two numbers to be added. These operands follow the instruction, on the same line. The instruction itself is like the verb of the sentence.

In addition the instruction has attributes. These are the permitted data sizes, which can be one or more of the types: byte, word or long word depending on the instruction. Also as a consequence of the instruction certain flags will be set or cleared in the condition code (status) register.

The list below gives the assembler mnemonics for the main instruction types.

Mnemonic	*Action*	*Mnemonic*	*Action*
ABCD	add decimal with extend	ADD	add
AND	logical and	ASL	arithmetic shift left
ASR	arithmetic shift right	Bcc	branch conditionally*
BCHG	bit test and change	BCLR	bit test and clear
BRA	branch always	BSET	bit test and set
BSR	branch to subroutine	BTST	bit test
CHK	check register against bounds	CLR	clear operand
CMP	compare	DBcc	test condition, decrement and branch*
DIVS	signed divide	DIVU	unsigned divide
EOR	exclusive OR	EXG	exchange registers

EXT	sign extend	JMP	jump
JSR	jump to subroutine	LEA	load effective address
LINK	link stack	LSL	logical shift left
LSR	logical shift right	MOVE	move
MOVEM	move multiple registers	MOVEP	move peripheral data
MULS	signed multiply	MULU	unsigned multiply
NBCD	negate decimal with extend	NEG	negate
NOP	no operation	NOT	ones complement
OR	logical or	PEA	push effective address
RESET	reset external devices	ROL	rotate left with extend
ROR	rotate right with extend	ROXL	rotate left with extend
ROXR	rotate right with extend	RTE	return from exception
RTR	return and restore	RTS	return from subroutine
SBCD	subtract decimal with extend	Scc	set conditional*
STOP	stop	SUB	subtract
SWAP	swap data reg. halves	TAS	test and set operand
TRAP	trap	TRAPV	trap on overflow
TST	test	UNLK	unlink

*A list of condition codes is shown below:

Condition Codes

CC	carry clear	CS	carry set
EQ	equal	F	false (never true)
GE	greater or equal	GT	greater than
HI	high	LE	less or equal
LS	low or same	LT	less than
MI	minus	NE	not equal
PL	plus	T	always true
VC	no overflow	VS	overflow

The condition codes follow instructions such as DBcc and Bcc, but be careful! The codes test the result of a calculation in the order

(destination operand) - (source operand),

placing the result (if any) in (destination).

DBcc (which is used for loop processing) will go to the next instruction if the condition is true, whereas Bcc (used for a straight branch) will branch if the condition is true (and go to the next instruction if it is false).

The most obvious loop instruction DBRA (decrement a counter and branch until it is -1) is actually absent from the 68000 set. But instead DBF (decrement and branch, never true) achieves the same result. Most assemblers implement DBRA anyway (but convert it to DBF on assembly), as a service to mankind.

Variations of Instruction Types

Here are additional variations of the main types. Most important are the endings -Q and -I which refer to faster "Quick" and "Immediate" versions; Quick being the faster of the two.

ADDA	add address	ADDQ	add quick
ADDI	add immediate	ADDX	add with extend
ANDI	AND immediate		
ANDI to CCR	AND immediate to cond. code		
ANDI to SR	AND immediate to status reg.		
CMPA	compare address	CMPM	compare memory
CMPI	compare immediate		
EORI	exclusive OR immediate		
EORI to CCR	exclusive OR immediate to condition codes		
EORI to SR	exclusive OR immediate to status register		
MOVEA	move address	MOVEQ	move quick
MOVE to CCR	move to condition codes		
MOVE to SR	move to status register		
MOVE from SR	move from status register		
MOVE to USP	move to user stack pointer		
NEGX	negate with extend		
ORI	OR immediate		
ORI to CCR	OR immediate to condition codes		
ORI to SR	OR immediate to status register		
SUBA	subtract address	SUBI	subtract immediate
SUBQ	subtract quick	SUBX	subtract with extend

Appendix 2

Devpac Assembler

There are many good assemblers available. The Devpac ST Assembler/Debugger by Hisoft has been used to develop the programs in this book because it is powerful, friendly and popular; there are many commands available. What is included in this appendix is the small subset which has been found to be especially useful.

The more recent version, DevpacST version 2, provides the option of editing, assembling, running and debugging a program all within the one environment. This gives the speediest development of programs. But the earlier version 1, which edits and assembles separately from debugging, is still entirely adequate for the job. The notes here apply mostly to version 2. If version 1 is used it will be necessary to delete the section headings SECTION TEXT, SECTION DATA and SECTION BSS (or precede them with an asterisk, *) from the main control programs. In addition, version 1 only allows assembly to a binary file on disk, whereas version 2 allows assembly to memory, within the Editor. This difference only matters when the GEMDOS call `#0, TERMINATE,` is invoked. In these programs, control will return to the Editor in version 2 but not to the Desktop in version 1. In the latter case the computer must be reset by turning it off for 10 seconds.

One last general point. It helps to have a high resolution monitor (SM124) available for program development; the Editor displays a complete page and the Debugger gives a more extensive view of memory contents. In low resolution less is visible on the screen.

GENST

This is the combined editor, assembler and debugger. You can write programs, run and debug them all within GENST2.

The Editor

This is a friendly screen editor, allowing you to roam freely through the entire program. Tabs can be set to convenient column positions in the instruction line which will consist of the following fields separated by spaces:

```
label       mnemonic     operand(s)       comment
```

The label is actually an address in RAM though it appears in the program as a user-friendly word, usually having a meaning which is relevant to the program. For example if it is the point to which the program returns in a repetitive loop, it might be simply "loop". Instruction mnemonics and operands have been discussed in Appendix 1. The comment field should explain in an informative way what is going on so that the progress of the program can be easily understood. An example might be

```
loop        move.w    d0,(a0)    save the flag
```

Moving About the File

Gross movements about a file are easily done by using the mouse to drag the slider on the scroll bar. To go to the start (top) or end (bottom) of a file press Alternate-T or Alternate-B, respectively.

The cursor keys can be used to control movement within the screen.

Editing Text

Whole lines can be deleted by pressing Control-Y, and restored by pressing Control-U (useful for repeating lines). Deleting within a line can be done by pressing Backspace (backwards) or Delete (forwards).

Text Movement

Among the most useful facilities are those which handle blocks of text. First move the cursor to the start of the block and press F1. Go to the end of the block and press F2. A marked block can be manipulated in several ways (Help lists these):

F3 saves a block; F4 copies it (to where the cursor is),

Shift-f4 saves it to the block buffer (a speedier version of F3),

Shift-F5 deletes it (but also saves it in the block buffer!),

F5 pastes in the block (at the cursor).

Alternate-W prints it out.

Assembly

A program can be assembled in several ways. Just to see whether it will assemble choose the Output to None option. This is the best thing to try on the first attempt. To run and debug a program choose the Output to Memory option. To save the assembled program to run independently choose the Output to Disk option and name it with the file extension .PRG (or TOS).

Options

There are many options available which affect how the assembly should take place. The option OPT-D (written at the top of the source file but after a BRA to the actual program) is very useful and will retain labels in the debugger, which helps enormously to follow the program.

Directives

Assembler directives, which have a similar appearance to assembler instruction mnemonics but which are unique to the assembler, are fairly standard. The common ones, such as EQU, DC, DS, used to fix the values of labels, set up (tables of) constants and to set up variables space, respectively, are used extensively throughout the example programs. Also used extensively to pull in files at assembly is the INCLUDE directive. This has made it possible to build up the book and the overall program by stages. The programs themselves show best how the directives are used.

Debugging

All assembly language programs have errors. Often, more time is spent debugging programs than writing them and so it helps to have a good debugger.

The debugger is actually called MonST and is available as a free standing program or within the Editor. Using it within the Editor makes the cycle of editing, assembling, running, debugging complete. Most likely you will want to single step through a program and watch what happens in the 68000 registers and in memory. Three windows display the register contents, a disassembled section of program around the current address of the program counter and the contents of a selected part of memory. A fourth small window passes messages. For the purpose of changing addresses and register contents, any one of the display windows can be made active by toggling Tab.

Executing Programs

There are many ways of monitoring a program. Here are some of them:

Control-Z	single step; every instruction executed
Control-T	single step; skips BSR's, JSR's, LineA, Traps
Control-A	single step; places a breakpoint after next instruction (useful for by-passing DBF's (DBRA's)
Run	produces a prompt for the type of run: G run at full speed to next breakpoint S run at reduced speed I run for (specified) count U run until condition is true (evaluate anexpression)

Breakpoints

These allow you to stop the program at specific addresses. They control the flow of the program in the different running modes. Here are simple controls:

Alternate-B	set a breakpoint at an address (and clear if followed by -)
Control-K	clears all set breakpoints
U	asks for an address to run to.
Help	show Help and breakpoints

Miscellaneous

Control-C	terminate MonST
L	list labels
P	print out (active window)
M	modify address
Alternate-A	set the starting address (active window)
Alternate-R	change contents of named register
Shift-Alternate-Help	interrupt running program

Hunting for Bugs

This is a skill learned through experience. The most useful tip is to check programs thoroughly before trying them. Try to construct programs in a structured way, in modules, each of which can be thoroughly tested independently before joining them all together. Do not rely on the Debugger to find the mistakes. By that time you'll have forgotten what each part of the program was for. Don't be in a hurry; don't spend one hour "bugging" and ten hours debugging!

A most common error is a bus error. This is when the program counter finds itself pointing to a wrong part of memory. This is often caused by the Stack getting out of order, particularly when a return address from a subroutine is required. Look to see how you have been using the Stack during the subroutine.

Appendix 3

Number Systems

Binary

Computers are made from electronic switches which are either off (0) or on (1). The number system which can be constructed out of such units is called binary (base 2), meaning out of 2; the system which goes in powers of 10 is called denary (base 10). In the binary system numbers are assembled from powers of 2. For example:

$$13(\text{base } 10) = 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0$$

Instead of writing numbers out in this long form it is usual to arrange only the coefficients of the powers of 2 in columns. The column number, labelled from the right, gives the power of 2. Hence the number 11 is written as

$$13_{10} = 1011_2$$

Each one of the units in the binary number is called a binary digit, or bit for short. The group of four bits is called a ''nibble'', especially loved by assembly language programmers who have frequent use of it.

A group of 8 bits also has a special name, a ''byte'', whose common use largely dates from the age of 8–bit microcomputers, which transferred data in bytes. In more recent 16–bit microprocessors (this microprocessor labelling scheme refers to the size of the data bus) such as the 68000, groups of 16 and 32 bits are commonly used. these are called ''words'' and ''long words'' respectively.

Hexadecimal (hex for short)

Humans count in powers of 10 (probably because they have 10 fingers), and find it unnatural to count in powers of 2. But some link with the binary system is necessary for assembly language programmers, especially when memory locations are being inspected. To this end the hexadecimal number system is commonly used. In it nibbles are abbreviated into single symbols. For the values up to 9 ordinary denary numbers are used but for the values 10 to 15 (the maximum value of a nibble) new symbols are needed. Here a great opportunity has been lost. Instead of inventing new computer age symbols, the letters of the alphabet A, B, C, D, E, F have been hijacked. Hexadecimal means base 16.

In the three systems binary, denary and hexadecimal respectively, the equivalence is:

Binary	Denary	Hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Negative Numbers

Negative numbers in binary are hard to get the hang of. This is because there is no special symbol reserved for the minus sign and it must be encoded within the number itself. It is done in the following way.

For simplicity, suppose we are working only in nibble size numbers (in fact there aren't any instructions to handle only numbers of this size on the 68000, a nibble must be part of a larger number). To deal in negative numbers the total possible range, 0–15, is split equally. The interval 0–7 inclusive (8 numbers) is reserved for positives and the range 15–8 inclusive (also 8 numbers) is reserved for negatives (the range –1 to –8). It's not as daft as it sounds. A negative number is obtained by

counting backwards from 0. If there is nothing below 0 the next best to do is to go to the top and count down. In a practical sense this is a good method because all the negative numbers have their top bit set. The top bit is like a minus sign turned vertical. There is a fancy name for this convention: 2's complement

There is a simple recipe for getting the negative of a number: write it in binary, switch all the 1's to 0's and 0's to 1's and then add 1. Let's try it. We know that –2 is in fact 14 so here's the check:

Step 1

+2 is	0010

Step 2 (2's complement)

change bits	1101
and add 1 to give	1110

which is 14 and therefore correct.

The 2's complement method of labelling negative numbers works for any size: bytes, words and long words. But be warned, only you know that the number is –2 and not 14, the computer doesn't! To help you keep track of what is going on the 68000 has instructions, called signed instructions which treat the top bit as a sign bit. There are other, unsigned instructions, which treat numbers as positive only. These help, but there are many occasions where the programmer must watch that numbers do not exceed their allotted range and flip sign, usually with pathological consequences.

In assembly language the different number types are distinguished by their different prefixes:

denary – none ; binary – % ; hex – $.

Appendix 4

ST Operating System

As they say at the start of "A Hitchhiker's Guide to the BIOS" (1985 Atari Corp.): **DON'T PANIC.**

The Operating System of the ST is big and complex. It is called TOS. It is basically split into two parts: one which depends on the hardware details of the ST and is called machine dependent (BIOS, XBIOS and Line A routines), and another which is machine independent (BDOS, XBDOS, VDI and AES) and will work on any computer which runs on the same operating system. A crude breakdown of their functions is

BDOS and XBDOS	basic disc operating system
VDI	graphics, particularly input
AES	graphics, particularly output
BIOS	basic input and output to all devices
XBIOS	bios extension to mouse, joystick sound and screen
Line A	very fast graphics primitives to screen.

Calls to the Operating System

BDOS — Push parameters onto the stack in the given order. Push the function number and call trap #1. Afterwards the stack must be tidied (the pointer returned to its precall value). Any returned parameter will be in D0.

XBDOS — An address pointer (to a parameter block) is placed in D1, the function code in D0 and trap #2 is called. A returned parameter (if any) is passed in D0. Access to AES and VDI is through this route.

BIOS Push parameters onto stack and call trap #13. Reply is passed in D0. Tidy stack.

XBIOS Push parameters onto stack and call trap #14. Reply is passed in D0. Tidy stack.

LINE A Declare constant $A000. Set up parameter blocks. Declare constant $A00n, where n is the routine to be called.

The Line A functions are described in greater detail in Appendix 5.

The following describes some of the Operating System calls that have a relevance to the programs in this book. The reader who wants more detail should consult the references given at the end.

BIOS calls (trap #13)

1 - bconstat (return character_device input status)

push: WORD bconstat; WORD character_device number. tidy #4
Returns character_device input status:
D0.L = $0000 if none; $ffff if some characters waiting.

device can be one of:
0 - prt (parallel port printer); 1 - aux (aux, RS232 port)
2 - con (console,the screen); 3 - midi; 4 - keyboard port;
5 - raw console (to screen without control)

Other functions also use character_device so a table can be made:

operation	prt	aux	con	midi	kbd	raw
bconstat	no	yes	yes	yes	no	no
bconin	yes	yes	yes	yes	no	no
bconout	yes	yes	yes	yes	yes	yes
bcostat	yes	yes	yes	yes	yes	no

The keyboard device is output-only, and can be used to configure the intelligent keyboard (ikbd) (or drive it insane).

2 - bconin (input character from device - wait for it)

As ***bconstat.***

Waits for a character to be input. Returns with code in low word of D0.L. For the console (dev. 2) returns the IBM-PC compatible scan code in low byte of upper word and the ASCII character in the low byte of low word (see Appendix 8).

3 - bconout (output character to device)
push: WORD bconout; WORD device; WORD character. tidy #6
returns when the character has been written (see Appendix 8).

8 - bcostat (return character device output status)
push: WORD bcostat; WORD device. tidy #4
returns status: 0 = not ready; -1 = ready to send

XBIOS calls (trap #14)

2 - _physbase (get address of physical screen (start of 32K block))
push WORD _physbase. tidy #2
returns address in D0.L (at next vertical blank)

3 - _logbase (get address of logical screen; right away)
push WORD _logbase. tidy #2
returns immediately the address in D0.L

4 - _getrez (get the screen's current resolution (0, 1 or 2)
push WORD _getrez. tidy #2
returns result in D0.W

5 - _setscreen (set up all (or some) of screen parameters)
push: WORD rez; LONG physbase; LONG logbase;
WORD _setscreen
tidy #12
(negative parameters are ignored allowing partial changes)

6 - _setpalette (set up the hardware colour palette)
push: LONG palettePtr; WORD -_setpalette. tidy #6
set the contents of the hardware palette register (all 16 entries)
from the 16 words pointed to by _palettePtr (at next Vblank).

7 - _setColour (set a colour register in the colour palette)
push: WORD colour; WORD colourNum; WORD _setColour. tidy #6
set the colour number ''colourNum'' in the palette to the value
''colour''. return old value in D0.W, no change if negative.

25 - ikbdws (write a string to the intelligent keyboard)
push: LONG ptr; WORD cnt; WORD ikbdws. tidy #8
ptr is the pointer to the string, cnt is the number of characters minus 1.

34 - kbdvbase (find the vector table for intelligent keyboard)
push WORD kbdvbas. tidy #2
returns in D0.L the pointer to the base (kbdvbase) of the table of vectors to ikbd routines called packet handlers that process the data packets received from the intelligent keyboard controller. The structure of long words is:

midivec	midi input
vkbderr	keyboard error
vmiderr	midi error
statvec	ikbd status packet
mousevec	mouse packet
clockvec	clock packet
joyvec	joystick packet
midisys	system MIDI vector
ikbdsys	system IKBD vector

The user can substitute his own vectors in this table so as to provide mouse and joystick control of programs.

VT52 Terminal Escape Codes

Text can easily printed on the screen using the VT52 terminal emulator routine in the BDOS. The program `file joy_test.s` in Chapter 8 illustrates this. Here are the control codes, each one of which must be preceded by the ESCAPE code 27:

CODE	*FUNCTION*	*DESCRIPTION*
A	cursor up	move cursor up a line (no effect at top)
B	cursor down	move cursor down a line (no effect at bottom)
C	cursor right	move cursor to right one column
D	cursor left	move cursor to left one line
E	clear home	erase screen and return cursor to (0,0) (top LH)
H	home cursor	return cursor to (0,0) (top LH)
I	cursor up	move cursor up a line (scrolls at top)
J	clear below	the screen below the cursor is erased
K	clear to eol	erase to end of line
L	insert line	insert blank line above present one
M	delete line	erase current line and close up space
Y,m,n	position at m,n	position cursor at row m, column n
b,f	foreground f	select character colour f
c,b	background b	select background colour b

d	clear to cursor	erase screen to cursor, inclusive
e	show cursor	make cursor visible
f	hide cursor	make cursor invisible
j	save cursor	save cursor position
k	restore cursor	restore it to saved position (or (0,0))
l	erase line	keep the current line but erase characters
o	clear to sol	erase to start of line, inclusive
p	reverse video	exchange foreground and background colours
q	normal video	restore fg and bg colours
v	wrap on	continue text onto next line
w	wrap off	characters pile up in last column position.

References

The Concise Atari ST Reference Guide. K.D.Peel, Glentop Press

Atari ST Internals. A Data Becker Book, Abacus Software

A Hitchhiker's Guide to the BIOS, Atari Corp.

Appendix 5

Line A (A–Line) Routines

The ST has a set of fast graphics routines available in User or Supervisor mode. In User mode they are triggered by any instruction which has the hexadecimal form $A00n where n is the number of the routine which can be from 0 to $F. Any instruction which begins with $A000 triggers an exception which is a special design feature of the 68000 incorporated to handle interrupts and exceptions of various kinds. This so called Line A "emulator" is to allow the user to set up customised routines which do not have to be called as subroutines.

When the code is detected, the processor switches to supervisor mode and selects the appropriate exception handler routine address from the list of addresses (jump vectors) at the start of RAM and is directed to the particular routine to execute according to the value of n. When the routine is completed, control again returns to the user at the instruction following the Line A code. Prior to calling any of the other Line A routines the very first one, $A000, must be called as an initialisation in order to find the addresses of parameter blocks, which then have to be set up before the routine is called.

The following details of the Line A routines are taken from S.A.L.A.D. *(Still Another Line A Document)* made available by Atari Corporation, U.K. and used by their kind permission. The 15 opcodes are:

Initialization	$A001	Return Line A pointers
Put Pixel	$A001	Draw a pixel
Get Pixel	$A002	Return the value of a pixel
Arbitrary Line	$A003	Draw an arbitrary line
Horizontal Line	$A004	Draw a horizontal line
Filled Rectangle	$A005	Draw a filled rectangle
Filled Polygon	$A006	Draw a horizontal. line of filled rectangle

BitBlt	$A007	Move/copy a section of memory
TexBlt	$A008	Move text to the screen
Show Mouse	$A009	Show the mouse pointer
Hide Mouse	$A00A	Hide the mouse pointer
Transform Mouse	$A00B	Transform the mouse pointer
Undraw Sprite	$A00C	Undraw software "sprite"
Draw Sprite	$A00D	Draw software "sprite"
Copy Raster	$A00E	Copy raster form
Seedfill	$A00F	Seedfill

Initialization $A000

This constant must be declared before any other. It returns several useful pointers including the one to the Line A Variables Structure. Returned registers contain:

d0 – pointer to variables structure
a0 – ditto
a1 – pointer to null terminated array of pointers to system font headers letting you to point to custom fonts in TexBlt call.
a2 – pointer to null terminated array of pointers to Line A routines so they can be called directly in supervisor mode.

This routine, like all the others is called by declaring the constant

```
dc.w $A000
```

This only needs to be done once so that the pointers can be stored for later use.

Line A Variable Structure

After initialization ($A000) both D0 and A0 point to the start of the variables structure. This is a long table into which the user must enter values before calling other Line A routines. Some of the entries contain pointers to secondary variables structures. The list is very long. Most of the lower address variables are concerned with the more complicated functions beyond $A007. Only the part concerned with simple graphics is given here. The variables are listed in order of their addresses, given as an offset from the base address returned by $A000.

Variables Concerned with Simple Graphics

NAME	*OFFSET*	*SIZE*	*FUNCTION*
PLANES	+000 $000	word	number of bit planes in current resolution
WIDTH	+002 $002	word	width of destination memory form in bytes; low and medium res: $a0(160 decimal); high res: $50 (80 decimal)
CONTRL	+004 $004	long	pointer to CONTRL array
INTIN	+008 $008	long	pointer to INTIN array
PTSIN	+012 $00C	long	pointer to PTSIN array
PTSOUT	+020 $014	long	pointer to PTSOUT array
COLBIT0	+024 $018	word	current colour bit value for colour plane 0
COLBIT1	+026 $01A	word	ditto 1
COLBIT2	+028 $01C	word	ditto 2
COLBIT3	+030 $01E	word	ditto 3 these are the four bits of the colour nibble which selects one of 16 coloursfrom the palette
LSTLIN	+032 $020	word	used in line drawing;iif zero last pixel drawn: nonzero last point undrawnfor conflicts e.g. connecting lines in XOR mode
LNMASK	+034 $022	word	
WMODE	+036 $024	word	equivalent to the VDI writing mode 0–replace, 1–transparent, 2–XOR mode, 3–reverse transparent
X1	+038 $026	word	x1 coordinate start of line
Y1	+040 $028	word	y1
X2	+042 $02A	word	x2 coordinate end of line
Y2	+044 $02C	word	y2
PATPTR	+046 $02E	long	pointer to fill pattern e.g. in horizontal line and filled rectangle
PATMSK	+050 $032	word	"mask" for fill pattern,maintains alignment of pattern on the screen,is ANDed with Y1 to give offset into pattern,most often is length of pattern minus 1,usually pattern is power of two in length
MFILL	+052 $034	word	multi–plane fill flag, 0 – fill pattern is single plane nonzero – multiplane
CLIP	+054 $036	word	clipping flag: zero = no clipping, nonzero = clipping
XMINCL	+056 $038	word	minimum x clipping value
YMINCL	+058 $03A	word	minimum y
XMAXCL	+060 $03C	word	maximum x
YMAXCL	+062 $03E	word	maximum y

Line A Routines

$A000 – initialization

Returns pointers to variables arrays
INPUT: none
RETURNS: D0 = pointer to variables structure base address
A0 = ditto
A1 = pointer to null terminated font headers
A2 = pointer to null terminated array of Line A pointers

$A001 – Put Pixel

Plot a single pixel
INPUT: INTIN[0] = pixel colour
PTSIN[0] = x coord. of pixel
PTSIN[1] = y coord. of pixel
RETURNS: nothing

Set up the INTIN and PTSIN arrays as shown above and then load their addresses at 8(A0) and C(A0) respectively.

$A002 – Get Pixel

Gets the value of a single pixel
INPUT: PTSIN[0] = x coordinate of pixel
PTSIN[1] = y

RETURNS: value of pixel in DO

Set up the arrays and pointers to the arrays as in $A001

$A003 – Arbitrary line

Draws line between two points
INPUT: COLBIT0, COLBIT1, COLBIT2, C0LBIT3, LISTLIN,
LNMASK, WMODE
X1,X2,Y1,Y2.
RETURNS: nothing

LINMASK is rotated to align with the rightmost end point.

$A004 – Horizontal line

Draw a horizontal line. Slightly faster than $A003
INPUT: COLBIT0, COLBIT1, COLBIT2, COLBIT3, WMODE, X1,
Y1, X2,PATPTR, PATMSK, MFILL
RETURNS: nothing

PATPTR points to an array of line patterns Which one is chosen depends on YI and PATMSK. If MFILL is nonzero, all planes will be filled with the values in the colour bits. This overrides WMODE.

$A005 – Filled rectangle

Draw a filled rectangle between the limits of X1, X2 and Y1, Y2.

INPUT: COLBIT0, COLBIT1, COLBIT2, COLBIT3, WMODE, X1, X2, Y1, Y2,PATPTR, PATMSK, MFILL, CLIP, XMINCL, XMAXCL, YMINCL, YMAXCL

RETURNS: nothing

The pattern length, PATMSK, should be 1 less than the length of the number of words in the pattern block.

$A006 – Filled polygon

This is not a substitute for fast polygon filling routines since it fills only one line at each call of $A006. To fill the entire polygon the value of y must be incremented and entered at Y1 in a loop.

INPUT: PTSIN[] (the array must be filled with the list of wordlength polygon vertices with the first one repeated at theend: x1,y1,x2,y2,.....xn,yn,x1,y1)
CONTRL[1] (number of vertices)COLBIT0, COLBIT1, COLBIT2, COLBIT3, WMODE, Y1 (current scanline y value), PATPTR, PATMSK, MFILL, CLIP, XMINCL, XMAXCL, YMINCL, YMAXCL.

RETURNS: nothing

$A007 – BitBlt

Perform a BIT BLock Transfer. This can be used to create a 'sprite'. It transfers a prepared bit pattern from one part of memory (called the source form memory) to another (called the destination form memory). The destination memory may be the screen, or some other part of RAM. In these memory 'spaces' the block of pixels to blit is located by the coordinates of its top left–hand corner (minimum x and y) called the anchor point and the size by its height and width. The BitBlt routine has its own dedicated parameter block into which data must be loaded before the call. The block must be 76 bytes long with the last 24 bytes being kept free for use by the routine itself. The pointer to this parameter block must be loaded into A6 before the routine is called. Variables marked (D) may be destroyed during the blit. Further explanations of the meanings of some of the variables are given at the end.

B_WD	+00 $00	word	width of block to blit in pixels
B_HT	+02 $02	word	height of block to blit in pixels (D)
PLANE_CT	+04 $04	word	number of consecutive planes to blit (D)

FG_COL	+06 $06	word	foreground colour (logic op index: hi bit) (D)
BG_COL	+08 $08	word	background colour (logic op index: lo bit) (D)
OP_TAB	+10 $A	long	logic ops for all fore and background combos
S_XMIN	+14 $E	word	minimum x: source
S_YMIN	+16 $10	word	minimum y: source
S_FORM	+18 $12	long	source form base address
S_NXWD	+22 $16	word	offset in bytes to next word in line
S_NXLN	+24 $18	word	offset in bytes to next line in plane
S_NXPL	+26 $1A	word	offset from start of current plane to next plane
D_XMIN	+28 $1C	word	minimum x: destination
D_YMIN	+28 $1C	word	minimum y: destination
D_FORM	+32 $20	long	destination form base address (screen address)
D_NX_WD	+36 $24	word	offset in bytes to next word in line
D_NXLN	+38 $26	word	offset in bytes to next line in plane
D_NXPL	+40 $28	word	offset from start of current plane to next plane
P_ADDR	+42 $2A	long	address of pattern buffer (0=no pattern)
P_NXLN	+46 $2E	word	offset in bytes to next line in pattern
P_NXPL	+48 $30	word	offset in bytes to next plane in pattern
P_MASK	+50 $32	word	pattern index mask
SPACE	+52 $34	24 bytes	workspace required by routine

NOTES

The prefixes S_ and D_ refer to the bit pattern in memory and when it is copied onto the screen respectively.

S_FORM is the pointer to the block of memory defining the sprite and D_FORM is the pointer to the base address of the screen. These addresses must be on word boundaries (start at the next word). S_NXWD and D_NXWD are the offsets, in bytes, to the next word in a particular colour plane (see Chapter 2) for the source and destination and are: monochrome = 2, medium = 4, low res. = 8. S_NXLN and D_NXLN are the number of bytes between each y value which are 80 in high resolution (monochrome) and 160 in medium and low resolution. S_NXPL and D_NXPL are the number of bytes between colour planes. On the ST screen this value is always 2, but it could be different for the source.

There is no clipping in this routine so care must be taken to ensure the blit does not spill outside the destination memory space.

Logic Operations

There is a complicated set of logic operations associated with the blitting process which combine the pixels about to be copied (source) with those already there (destination). What logic operation applies to each colour plane is calculated from the contents of OP_TAB (see below).

This routine is complicated and powerful due to the variety of different options available. Its basic function is to move bit images defined within a rectangle from one part of memory to another. If the destination is the screen then it provides a very versatile framework within which to manipulate 2D pictures or 'sprites'.

There are other Line A functions ($A00C and $A00D) dedicated to handling small sprites of size 16x16 pixels but there is no such limit on $A007. They do however take care of the housekeeping associated with preserving the underlying image when the sprite is removed. There are three principal entities involved in this function having similar sets of variables but with different prefixes: source, destination and pattern. They represent the transfer of an image from the source to the destination with the possibility of including a pattern in the process.

There are many alternative settings of the variables to cope with the range of options available and ample opportunity for experimentation (which is probably the only way of finding out how it all works). Some of the more impenetrable are concerned with the way in which the foreground and background colours of the source interact with the destination especially when there are different numbers of planes in the source and destination. There are 16 logical operations, called RASTER OP codes, between the source, S, and destination, D, to give the following results:

OP CODE	*Combination Rule*
0	0
1	S AND D
2	S AND [NOT D]
3	S (replace mode)
4	[NOT S] AND D (erase mode)
5	D
6	S XOR D (XOR mode)
7	S OR D
8	NOT [S OR D]
9	NOT [S XOR D]
A	NOT D
B	S OR [NOT D]
C	NOT S
D	[NOT S] OR D
E	NOT [S AND D]
F	1

For each colour plane the logical operation is chosen from a table (at OP_TABLE) of 4 byte length codes which is indexed from the appropriate plane bits in the foreground colour (FG_COL) and background colour (BG_COL). The entries in this table depend strongly on how you want the foreground and background colours to interact with the destination The foreground bit is the high bit and the background bit is the low bit in a two bit number to index into the table.

Patterns

Patterns can be included in the blit unless the pointer, P_ADDR, is zero. The pattern is word–wide, an integral power of 2 in height (and vertically repeated at that spacing) and word aligned. Since the pattern is anchored to the coordinate (0,0) (upper left hand corner) of the destination memory form and is logically ANDed with the source prior to logical combination with the destination, the final pattern depends on the destination coordinates. P_NXLN is the offset in bytes (an integral power of 2) between consecutive bytes in the pattern.P_NXPL is the offset in bytes between consecutive pattern planes and a single plane pattern can be used to set all destination planes with the same pattern by setting the plane offset to zero. P_MASK works with P_NXLN to specify the length of the pattern which (in words) must be an integral power of 2. The relation between these two is

if P_NXLN = 2**n
then P_MASK = (length in words–1)<n.

Some Examples

1. To BLT from a single plane source to a multiplane destination set S_NXPL=0. The same source plane is BLTed to all destination planes. To map 1's to foreground colour and 0's to background colour set OP_TAB to:

Offset	*logic*	*Op*
+00	00	all zeros
+01	04	[NOT S] AND D
+02	07	S OR D
+03	15	all ones

To map 1's to foreground colour and make 0's transparent set S_NXPL to zero and OP_TAB to:

+00	04	[NOT S] AND D
+01	04	
+02	07	S OR D
+03	07	

2. To BLT a pattern without Source to the Destination, define a word of 1's and set S_FORM to point to it. Set S_NXLN, S_NXPL, S_NXWD, S_XMIN, and

S_YMIN to zero. Set up the pattern and the BLT will create a pattern filled rectangle.

3. To create a simple sprite–like device, build a monoplane mask. Everywhere there is a 1 in the mask the background will be removed. Everywhere there is a 0 the background is left intact. Set OP_TAB to:

+00	04	[NOT S] AND D
+01	04	
+02	07	S OR D
+03	07	

It is not necessary to enter a background colour BG_COL. Take a monoplane form (or a multiplane form) and "OR" it (OP7) into the area that you just scooped out with the mask.

Example: BitBlit a monochrome invertebrate to the screen

```
move.w    #2,-(sp)     find the screen
          trap         #14
          addq         #2,sp
          move.l       d0,screen
          lea          blit,a6          pointer to parameter block
          dc.w         $a007            BitBlt

*BitBlit parameter block
blit:     dc.w         $0030            width of source in pixels
          dc.w         $0014            height of source in pixels
          dc.w         $0001            number of planes to blit
          dc.w         $0001            fg colour (logic op index: hi bit)
          dc.w         $0000            bg colour (logic op index: lo bit)
          dc.l         07070707         logic ops for all fg and bg combos
          dc.w         $0000            minimum X: source
          dc.w         $0000            minimum Y: source
          dc.l         slug             source form base address
          dc.w         $0002            byte offset to next word in line
          dc.w         $0006            byte offset to next line in plane
          dc.w         $0002            offset to next plane (in bytes)
          dc.w         $00ff            minimum X: destination
          dc.w         $0064            minimum Y: destination
screen    dc.l         $00000000        destination form base address
          dc.w         $0002            byte offset to next word in line
```

```
            dc.w        $0050           byte offset to next line in plane
            dc.w        $0002           offset to next plane (in bytes)
            dc.l        $00000000       address of pattern buffer
            dc.w        $0000           byte offset to next line in pattern
            dc.w        $0000           byte offset to next plane in pattern
            dc.w        $0000           pattern index mask


* working space
            dc.w        $0000,$0000,$0000,$0000
            dc.w        $0000,$0000,$0000,$0000
            dc.w        $0000,$0000,$0000,$0000


* the image
slug:
* $30 pixels/scanline, $14 scanlines
* monochrome mask (1 plane: background = 0, foreground = 1)
        dc.w    $0000,$0000,$0030,$0000,$0000,$0066,$0000,$0000
        dc.w    $006c,$0000,$0000,$00ce,$0000,$0000,$00cc,$0000
        dc.w    $0000,$0198,$0000,$0000,$03b0,$0000,$0000,$0770
        dc.w    $0000,$0000,$0760,$0000,$0000,$0ee0,$0000,$0000
        dc.w    $7fc0,$0000,$0003,$ffc0,$0000,$003f,$ffc0,$0000
        dc.w    $00ff,$ffe0,$0000,$1fff,$fff0,$01ff,$ffff,$fef0
        dc.w    $0fff,$ffff,$ff70,$1fff,$ffff,$ff80,$ffff,$ffff
        dc.w    $ffe0,$ffff,$ffff,$ffc0
* note: this program changes "constants" in "dc.w"'s; it would
* be better practice to use the "ds.w" directive in the bss section.
```

Appendix 6

Vectors and Matrices

Vectors and matrices go together. Whatever convention is chosen for vectors determines the convention for matrices.

Vectors

A vector is a concise way of specifying a position in space. The position is measured from a fixed position called the origin. Since space is 3-dimensional the position is determined by moving specified distances forward, sideways right and up from the origin (negative distances account for backward, left and down respectively). In mathematical language this means measuring all displacements in a Cartesian coordinate system. A position in space is then specified by the distances along the three axes at right angles one has to travel to reach it. The vector notation arises from the way this information is presented. If the displacements along the three axes to the point, P, are x,y and z respectively, then the vector **r** which stretches from the origin to P, as shown in Figure A6.1, can be expressed in vector notation as

$$\mathbf{r} = x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$$

It is common to write vectors (which have both size (magnitude) and direction) in boldface to distinguish them from ordinary numbers which have only size. Here **i, j** and **k**, called the unit or base vectors, are signposts pointing along the x, y and z axes and the term x**i** means "go a distance x in the direction of the x axis" and so on. They are vectors in their own right with size (magnitude) equal to unity.

Since **i, j** and **k** really serve only to distinguish the three components of the displacement, we could omit them from the scheme providing the order is retained. The three components can be included in order inside brackets ready for multiplication with matrices in the column vector notation

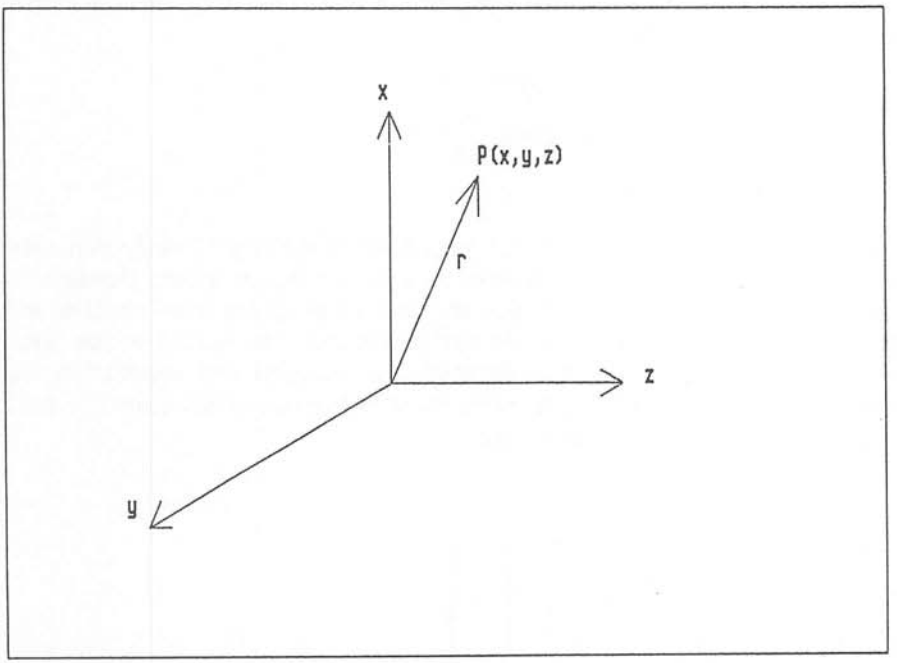

Figure A6.1 A vector in Cartesian coordinates

$$\mathbf{r} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

This is not the only way to represent vectors. In computer graphics it is common to represent them in the row notation

$$\mathbf{r} = (x\ y\ z)$$

The convention used determines the way matrices are written. In this book column vectors are used because this is more common in science and engineering and therefore likely to be more familiar to the general reader. Switching between the conventions is tiresome but fairly painless.

Matrices

As a result of rotational transforms which occur frequently in computer graphics, the coordinates of objects change in a particular way. A point P(x,y,z) will move to

a new position P′(x′,y′,z′) as a result of a rotation about some axis as shown in Figure A6.2. Each one of the new components is related to all the old components in a set of linear equations:

$$x' = M11.x + M12.y + M13.z$$

$$y' = M21.x + M22.y + M23.z$$

$$z' = M31.x + M32.y + M33.z$$

where the M's are numbers giving the proportions of the original components and are the elements of a matrix **M**. The important thing is that the matrix elements are related uniquely to the rotation, so that any other point rotated in an identical way about the same axis would have its new components determined by the same matrix **M**. Using the rules of multiplication of matrices and vectors, we can emphasise this by disentangling the elements of **M** from the components x, y and z of the vector. The product is written as:

$$\begin{matrix} x' = \\ y' = \\ z' = \end{matrix} \begin{pmatrix} M11 & M12 & M13 \\ M21 & M22 & M23 \\ M31 & M32 & M33 \end{pmatrix} . \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

The matrix product written this way is just shorthand notation for the set of linear equations which really matter when we actually come to work out the new coordinates. But writing it this way makes it clear that, once calculated, the matrix M can be used to rotate any point in the same way. In an even more concise shorthand we can summarise the transformation by:

$$\mathbf{r}' = \mathbf{M}.\mathbf{r}$$

where the product here is the matrix product and not an ordinary product of numbers.

To convert this shorthand product back into the set of equations observe that the vector has three rows and one column and the matrix has three rows and three columns. To form the top row (x′) of the transformed vector **r′**, multiply in turn each of the elements in the top row of **M** by each of the rows of the vector r and add them. The second row of **r′** is calculated from the product of each elements in the second row of M with the rows of r and so on (if we were working in the row representation of vectors everything would be the other way round). This meaning of matrix multiplication is something that just has to be learned.

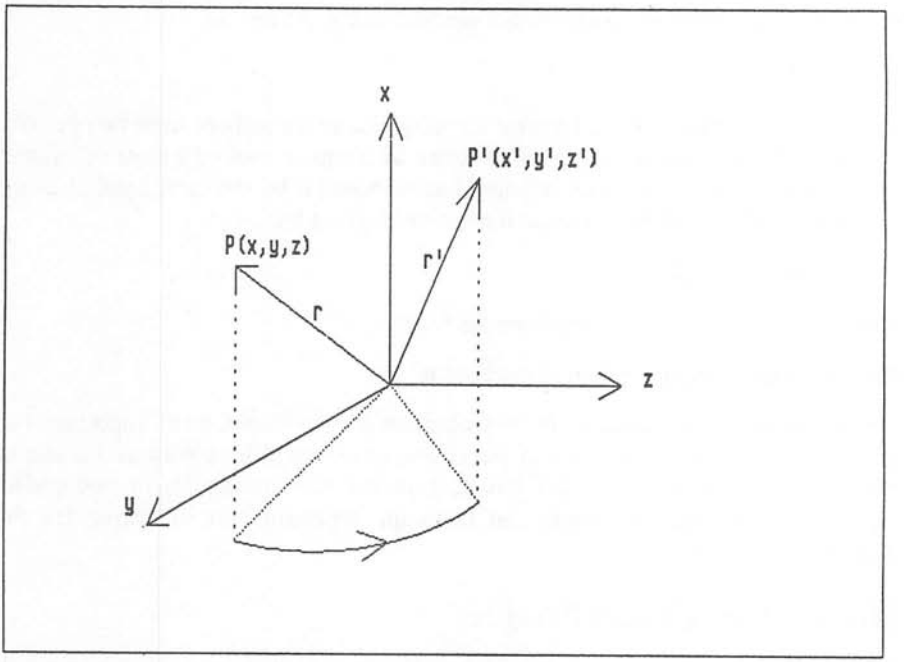

Figure A6.2 Point P transferred to point P'

Products of Vectors

The Scalar (Dot) Product

Vectors are really just a shorthand and highly suggestive way of doing geometry. A point P(x,y,z) in a Cartesian system looks much more important when represented by a vector **r** which stretches from the origin to the point P. Another point P' (x' ,y' ,z') is similarly represented by the vector P' .

Very often we wish to know the angle, θ, between these two vectors (referring back to the previous section it could be the angle of rotation of the vector P). It turns out that what is simplest to find is the cosine of θ which is

$$\cos\theta = (x.x' + y.y' + z.z') / \sqrt{((x^2+y^2+z^2).(x'^2+y'^2+z'^2))}$$

The factors in the denominator look complicated but are just the magnitudes of the two vectors calculated using a 3D version of Pythagoras' theorem. The numerator is the sum of the products of the components of the two vectors taken together.

Because such a product occurs frequently in geometry it is given a special symbol and name. It is called the scalar or dot product and is written as

$$\mathbf{r}.\mathbf{r}' = x.x' + y.y' + z.z'$$

It is called the scalar product because it produces a scalar answer from two vectors. Instead of writing the magnitude of a vector as a square root of a sum of squares all the time, which is tiresome, it is usual to represent it by the same symbol as the vector but without boldface. Hence the cosine is given by

$$\cos\theta = (\mathbf{r}.\mathbf{r}')/r.r'$$

where $r = |\mathbf{r}| = \sqrt{(x^2+y^2+z^2)}$ and likewise for $\mathbf{r}'$.

The operation $|\mathbf{r}|$ means 'the magnitude of **r**.'

Notice that the scalar product $\mathbf{r}.\mathbf{r}'$ is proportional to $\cos\theta$ and, most important, has the same sign as $\cos\theta$. The sign of the cosine turns out to be a very useful test of whether two vectors are parallel (pointing in the same direction) or antiparallel (pointing in opposite directions) and plays an important part in testing for the visibility of surfaces.

The Vector (Cross) Product

This is a product of two vectors which produces a new vector. Once again it is based on a useful application. In this case it generates the vector which is normal (at right angles) to both the original vectors. Another way of stating this is to say that the new vector is normal to the plane containing the two product vectors. This is shown in Figure A6.3. The new vector $\mathbf{r}''$ and the vector product are defined by:

$$\mathbf{r}'' = \mathbf{r} \times \mathbf{r}'$$

The vector $\mathbf{r}''$ is normal to the plane containing **r** and $\mathbf{r}'$ and its magnitude is equal to $r.r'.\sin(\theta)$. The components of $\mathbf{r}''$ are

$$x'' = y.z' - z.y'$$

$$y'' = z.x' - x.z'$$

$$z'' = x.y' - y.x'$$

There is one important aspect of vector products which is also true of matrix products, the order of multiplication matters; the product $\mathbf{r} \times \mathbf{r}'$ is not the same as $\mathbf{r}' \times \mathbf{r}$. In fact

$$\mathbf{r}' \times \mathbf{r} = -\mathbf{r} \times \mathbf{r}'$$

The direction of $\mathbf{r}''$ is obtained by twisting **r** into $\mathbf{r}'$ through the smallest angle. The direction in which this is seen as a clockwise rotation is the direction of $\mathbf{r}''$.

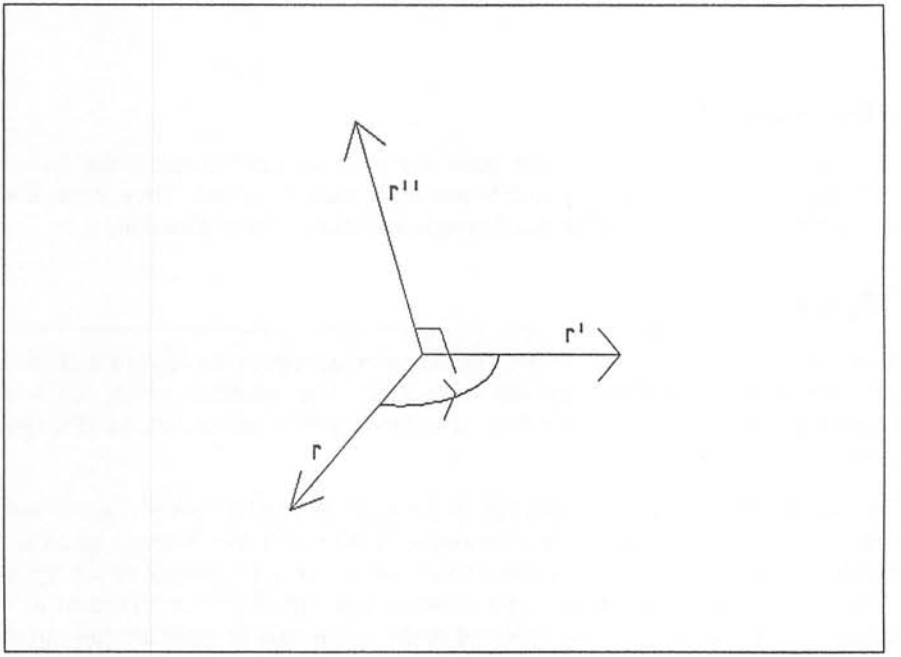

Figure A6.3 Vector cross product

The vector product is complicated but very useful in computer graphics. It is used to construct vectors which are normal to surfaces. We discuss this next.

Surface Normal Vectors

It is often necessary to construct a vector which is normal to two other vectors. This occurs in the calculation of surface normal vectors and coordinate transforms. In the case of a surface normal vector the objective is to construct a vector which is normal (at right angles) to the surface.

What this amounts to is forming the vector product of two vectors which lie in the surface, as discussed in the previous section. Usually these two vectors are not presented as such but have themselves to be constructed from polygon vertex coordinate lists. Suppose three consecutive vertices of a convex polygon are P1(x1,y1,z1), P2(x2,y2,z2) and P3(x3,y3,z3) and that these go clockwise round the perimeter. The two vectors which can be multiplied in a cross product to give a vector pointing out of the surface are

$$\mathbf{r} = (x3-x2)\mathbf{i} + (y3-y2)\mathbf{j} + (z3-z2)\mathbf{k}$$

$$\mathbf{r}' = (x2-x1)\mathbf{i} + (y2-y1)\mathbf{j} + (z2-z1)\mathbf{k}$$

so ,

$$\mathbf{r''} = \mathbf{r} \times \mathbf{r'}$$

Base Vectors

Base vectors are unit vectors which point along the axes of the coordinate system. In Cartesian coordinates, **i**, **j** and **k** are the "base" vectors. They each have magnitude 1, so the only thing that distinguishes them is their direction.

Matrices

Matrices have already been discussed in the previous section. In computer graphics they represent a transformation of some kind. The matrices which are most straightforward to deal with are those associated with rotation and are discussed further in Appendix 7.

The rule for multiplying two matrices in the same as that of multiplying a matrix and a vector (as discussed in the previous section) where the vector is taken as a matrix having one column and three rows. Adding extra columns to the vector makes it a matrix and produces extra columns in the product. For a product to be possible there must be as many columns in the first matrix as there are rows in the second matrix.

The matrices which describe rotation about the three axes x,y and z all have three rows and three columns (unless they are in homogeneous coordinates): they are 3x3 matrices. The act of building up a complex rotation from the separate matrices in some order is accomplished by multiplying the matrices together. This is called matrix concatenation. Just as with the vector cross product, the order of the matrix multiplication matters: the matrix farthest to the right is the first rotation and that closest to the left is the last rotation.

Homogeneous Coordinates

Unlike rotations, certain types of transform, such as translations and perspectives, cannot be written as 3 x 3 matrices and made to operate on vectors as a product. Since, for the purpose of concatenation, it is desirable to put all transforms on an equal footing, homogeneous coordinates are used to convert all transforms to 4 x 4 matrices which can be multiplied.

This means moving to a 4-D space (not real space, just a mathematical convenience) in which the additional dimension is always 1. The extra degree of freedom this gives is sufficient to convert all transforms to 4 x 4 matrices. Likewise all vectors must have a forth component, 1. Putting this fourth dimension to unity means we are working on a "plane" in the 4-D space which has the intersection 1. The "plane" is normal 3-D space.

Appendix 7

Geometric and Coordinate Transforms

There are two types of transform used widely in computer graphics: geometric and coordinate transforms. What is confusing is that they are really two aspects of the same thing and it is possible to achieve the same end result by either method. However in order to stay sane it helps greatly to think of them as different, chosing one or the other depending on the problem. Many clever shortcuts become possible once the distinction and connection between them is understood.

Imagine that you are sitting in a swivel chair positioned at the centre of circular carpet in a room with black featureless walls. Since there is no external reference point (apart from remembering what actually happened) it is not possible to distinguish between rotating the chair to the right on a stationary carpet, or keeping the chair fixed and rotating the carpet to the left. The observer on the chair sees the same relative movement of chair and carpet and his view of the carpet pattern is the same in both cases. But we must be careful to establish a scheme of rotation of either the chair or the carpet which are consistent. Let us decide that left rotations are positive and right rotations are negative. Then we can see that a positive rotation of the chair (the observer) is equivalent to a negative rotation of the carpet (the object): they are said to be the inverse of each other.

Now we come to the formal definitions. Rotating the observer is called a coordinate transform and rotating the object is called a geometric transform. There are many times in computer graphics when we wish to do both of these. When an object is moved in the world frame, it is subject to a geometric transform. When we wish to see the world from a different point of view a coordinate transform must be done. When the observer is controlling his viewpoint orientation by means of a joystick it is useful to exploit the connection between the two transforms.

Coordinate Systems and Frames of Reference

To some extent these terms are used interchangeably. For the most part the positions and vertices of objects are determined in Cartesian coordinates by a set of three x, y and z axes at right angles. The position of the zero of this set of axes is called the origin of the coordinate system. The whole constitutes a frame of reference to track subsequent motion of the various objects. As we have seen, there are two types of movement: a coordinate transform (when the observer moves) and a geometric transform (when an object moves). When the object moves it is easiest to keep track of what is going on by following the motion of the frame of reference attached to the object itself. We have called this the object frame. In the main text the object–to–world transform was made by selected rotations and a displacement of this object frame. Now we can see exactly how this works.

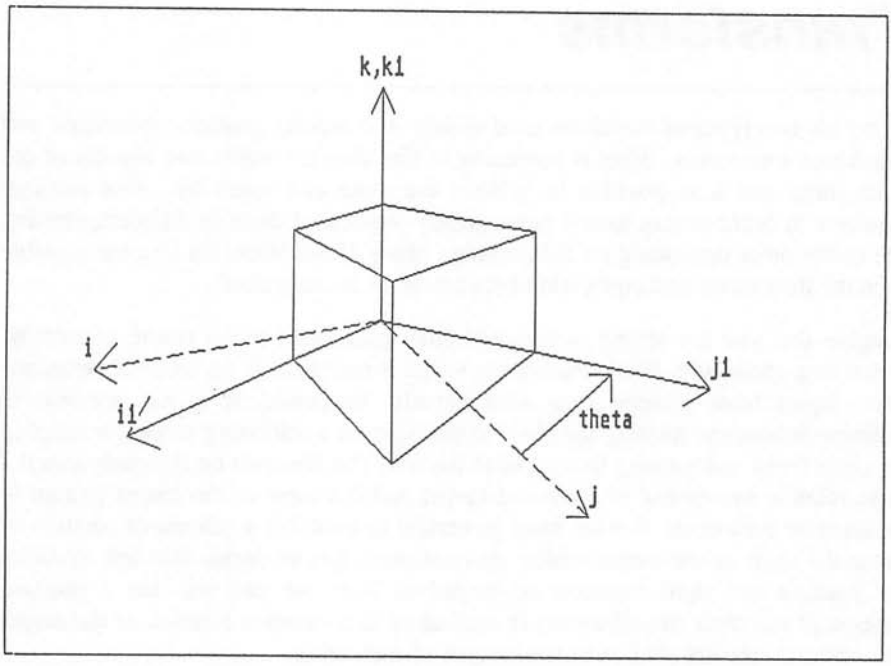

Figure A7.1 Rotation of an object

Imagine a set of axes permanently attached to the object so that when it moves they also move. For simplicity, we consider a rotation by an angle θ about the z axis, as shown in Figure A7.1. A transform matrix is now needed to relate the coordinates after the rotation (x1,y1,z1,) to those before (x,y,z). The beauty of this scheme is that we can construct this matrix by observing what happens to the base vectors. Remember, the base vectors are the unit vectors (of size 1) pointing like

sign posts along the x,y and z axes. The base vectors before the rotation are **i**, **j**, and **k** and after the rotation are **i1**, **j1** and **k1**.

Looking at the Figure we can see the relations between these:

$$\mathbf{i1} = \cos\theta.\mathbf{i} + \sin\theta.\mathbf{j}$$

$$\mathbf{j1} = -\sin\theta.\mathbf{i} + \cos\theta.\mathbf{j}$$

$$\mathbf{k1} = \mathbf{k}$$

leading to a transform matrix for the base vectors:

$$\begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Now this matrix as it stands cannot be used to transform the coordinates (x,y,z) to (x1,y1,z1), but curiously enough, its inverse can. Fortunately, the inverse of a pure rotation is simply obtained by switching (transposing) the rows and columns. In technical language, the inverse of a rotation is its transpose. Doing this yields the matrix:

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

so that, for example, in a rotation by 90 degrees, the point (0,1,0) becomes the point (0,0,1) and the point (0,0,1) becomes (0,–1,0). So we have found a way of rotating an object to a new orientation: perform that reorientation on the object base vectors and express the result in terms of the original base vectors; then transpose the matrix to produce the coordinate transform matrix.

Can the original matrix be used for anything? Yes. As it stands, before it is transposed, it is a coordinate transform. If we were to leave the object stationary and just rotate the frame of reference, it gives us the transform to calculate what the object coordinates appear to be in the new rotated frame. This is shown in

Figure A7.2. Hence in the rotation of 90 degrees, the vertex (0,1,0) appears to be at (0,0,–1), and the vertex (0,0,1) appears to be at (0,1,0) when seen from the rotated frame. Note that in both of these rotations, of the object and reference frame respectively, the sense of the rotation was positive.

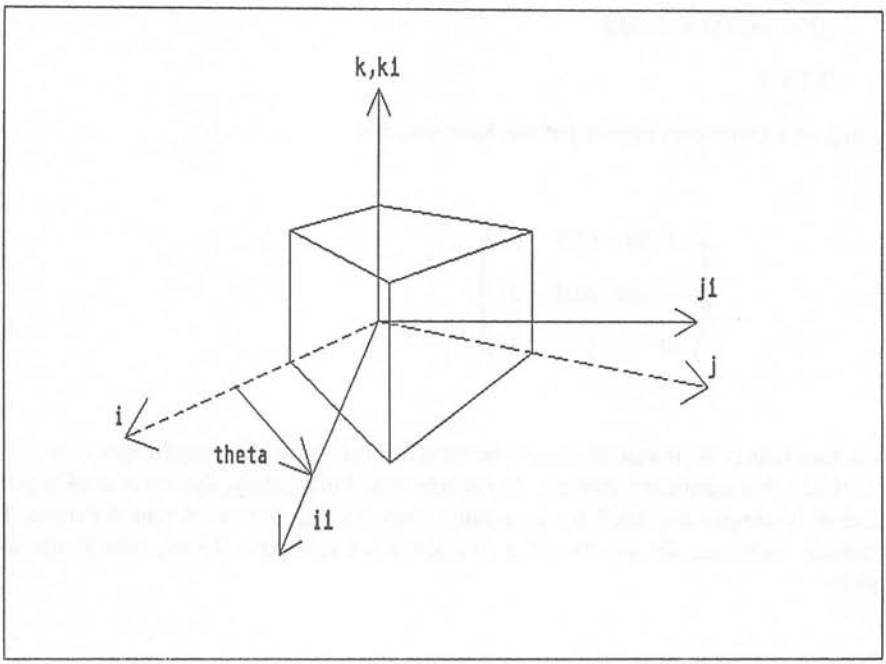

Figure A7.2 Rotation of a frame of reference

Now we can see the qualitative discussion concerning the observer on the swivel chair and the carpet expressed mathematically. The transform which calculates the coordinates of the object after its positive rotation is:

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

and the transform which calculates the new apparent coordinates of the stationary object after the reference frame has been moved in a positive direction is:

$$\begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

They are different when both involve a positive rotation but become the same if the reference frame (the chair) is rotated negatively. Then the angle θ is negative and because $\sin(-\theta) = -\sin\theta$ but $\cos(-\theta) = \cos\theta$ the terms involving $\sin\theta$ change sign but those involving $\cos\theta$ don't.

This is only restating the fact that rotating the reference frame one way gives the same relative motion as rotating the object the other way.

Appendix 8

Colour Palette and Key Scan Codes

Standard Colour Palette

RGB value (hex)	*Colour*
777	white
700	red
070	green
770	yellow
007	blue
707	magenta
077	cyan
555	light grey
333	grey
733	light red
373	light green
773	light yellow
337	light blue
737	light magenta
377	light cyan
000	black

GSX standard keyboard mapping

The key scan codes returned by the *ikbd* are chosen to simplify the implementation of GSX.

Hex	*Keytop*	*Hex*	*Keytop*	*Hex*	*Keytop*
01	Esc	24	J	47	HOME
02	1	25	K	48	UP ARROW
03	2	26	L	49	(NOT USED)
04	3	27	;	4A	KEYPAD
05	4	28	'	4B	LEFT ARROW
06	5	29	"	4C	(NOT USED)
07	6	2A	(LEFT) SHIFT	4D	RIGHT ARROW
08	7	2B	\	4E	KEYPAD +
09	8	2C	Z	4F	(NOT USED)
0A	9	2D	X	50	DOWN ARROW
0B	0	2E	C	51	(NOT USED)
0C	-	2F	V	52	INSERT
0D	==	30	B	53	DEL
0E	BS	31	N	54	(NOT USED)
0F	TAB	32	M	5F	(NOT USED)
10	Q	33	,	60	ISO KEY
11	W	34	.	61	UNDO
12	E	35	/	62	HELP
13	R	36	(RIGHT) SHIFT	63	KEYPAD (
14	T	37	(NOT USED)	64	KEYPAD)
15	Y	38	ALT	65	KEYPAD /
16	U	39	SPACE BAR	66	KEYPAD *
17	I	3A	CAPS LOCK	67	KEYPAD 7
18	O	3B	F1	68	KEYPAD 8
19	P	3C	F2	69	KEYPAD 9
1A	[	3D	F3	6A	KEYPAD 4
1B	]	3E	F4	6B	KEYPAD 5
1C	RET	3F	F5	6C	KEYPAD 6
1D	CNTL	40	F6	6D	KEYPAD 1
1E	A	41	F7	6E	KEYPAD 2
1F	S	42	F8	6F	KEYPAD 3
20	D	43	F9	70	KEYPAD 0
21	F	44	F10	71	KEYPAD .
22	G	45	(NOT USED)	72	KEYPAD ENTER
23	H	46	(NOT USED)		

INDEX

2's complement, 220
3D
    clipping in, 161
    Modelling, 26
    general transforms, 129
    order, 177
68000 Instruction Set, 209

A-Line routines, 226
Absolute addressing, 210
Absolute code, 32
Address registers, 209
Addressing modes, 209, 210
AES, 8, 221
Assembly language, 6, 32, 209, 214
Atari ST, 1
Atari STE, 26
Attributes, 176

Base vectors, 156, 242
BDOS, 221
BCD digits, 210
Binary, 209, 218
BIOS, 8, 9, 100, 221
BIOS calls (trap #13), 222
BITBLT graphics, 26, 26
Bits, 208, 210
BLITTER, 26
Blurring, motion, 5
Breakpoints, 217
Bresenham algorithm, 36, 38, 40
bss_00.s file, 41, 43, 50
bss_01.s file, 56, 64
bss_02.s file, 73, 80
bss_03.s file, 89, 99
bss_04.s file, 120
bss_05.s file, 139, 149
bss_06.s file, 164, 175
bss_07.s file, 187, 200

Bubble sort, 179, 179, 180, 186
Bugs, hunting for, 217
Bytes, 210, 218

Calls (to operating system), 221
Cartesian (x,y,z) coordinate system, 28, 30
Centre of projection, 66
Clip frame, 30, 51
clipfrme.s file, 55, 57
Clipping in 3D, 161
Clipping, 51, 55
Colour, 115
Colour Palette, 12, 115 - 119
Colour Palette, standard, 248
Colour planes, 13, 117
Column vectors, 32
Complex scene, 176
Composite rotations, 84
Computer Aided Design (CAD), 4
Computer games, 4
Computer graphics, 1, 2
Concatenation, 68, 129
Condition codes, 212
Control matrices, 152, 180 - 182
Control Panel Accessory, 117
Convex polyhedra, 52, 111
Coordinate systems, 30, 244
Coordinate transforms, 81, 132, 153, 243
core_00.s file, 42, 45
core_01.s file, 56, 58
core_02.s file, 73, 76
core_03.s file, 89, 90, 93
core_04.s file, 118, 123
core_05.s file, 138, 142
core_06.s file, 163, 167
core_07.s file, 186, 190
core_08.s file, 187, 207
Cosine tables, 88
Cosines, direction, 153, 156

crds_out, 56
Cross product (vectors), 113

Data
  packet, 102
  registers, 209
  structures, 32
Database, 176
data_00.s file, 16, 22
data_01.s file, 71, 79
data_02.s file, 71, 79
data_03.s file, 88, 92
data_04.s file, 119
data_05.s file, 139, 150
data_06.s file, 185, 201
data_07.s file, 182, 186, 203
data_08.s file, 185, 204
Debugging, 214, 216
Decision variable, 39
Depth sorting, 176
Devpac Assembler, 8, 214
Direction cosines, 153, 156
Directives, 216
Displacements, 129
Distant objects, 162
Dot product, 112

Edge list, 33
Editor, 214, 215
Effective address (ea), 210
Errors, 159
Euler angles, 151, 153, 156, 180, 185, 187
eulr_scn, 183, 184, 185, 206
Exceptions, 13
Executing Programs, 216

Fast filling, 36
Field of view, 162
Fields, 179
find_phys, 15
Fire button, 102
Flicker-free pictures, 87
Flight simulators, 29, 151, 153, 180, 184
Fractals, 3
Frame, 73
Frames of reference, 27, 244
Friction, 137
Frustrum of visibility, 161

GEM variables area, 104
GEMDOS, 8
General Transforms in 3D, 129
GENST, 214
Geometric transforms, 81, 127, 129, 153, 243
Geometry engines, 5, 26
Graphics primitives, 29
GSX standard keyboard mapping, 248

Hexadecimal, 219
Hidden surface removal, 3, 110, 111, 118
High resolution, 11
High resolution monitor, 214
Hisoft Devpac assembler, 1
Hither plane, 162
Homogeneous coordinates, 65 - 69, 86, 242
Horizontal blank, 12

IKBD, 100 - 104
IKBDWS, 103, 104
illkey, 118
Illumination, 110, 114, 115, 118
Illumination vector, 115
ill_hide, 118, 121
Immediate addressing, 210
Implied addressing, 210
INCLUDE directive, 6, 41
Independent code, 7
Indexed addressing, 33
Inertia, 137, 138
Initialization, $A000, 227
Input devices, 100
Instance transforms, 28, 129, 136
Instruction set, 68000, 209, 211
Instruction types, variations, 213
Intelligent Keyboard Controller, *see IKBD*
Interrupt, vertical blank, 87
INTIN, 14
Inversion, 129

Joystick, 100 - 102, 130
Joystick handler routine, 102
Joystick vector, 102
joy_test, 104, 108
Jump vectors, 323

KBDVBASE, 102
Key Scan Codes, 248
Keyboard mapping, standard GSX, 248
key_peek.s program, 104, 105

Labels, 8, 32, 33, 109
Latitude, 130

Left-handed Cartesian coordinates, 30
Light source, 110, 115
Line A (A-Line) Routines, 8, 9, 13, 14, 226, 229
Line A variable structure, 227
Line-of-sight vector, 114, 119
Linear transform, 68
LINK, 73
Lists, 33
Logical screen, 10, 29, 51, 87
Long words, 210, 218
Longitude and Latitude, 130
Look up tables, 15
Low resolution, 11, 12, 116
Low resolution screen driver, 15
Lucasfilm, 5

Map, 176
Mass, 137
Matrices, 31, 321, 236, 237, 242
Matrix concatenation, 242
Matrix product, 68
Medium resolution, 11, 13
Micropolygons, 5
Modelling, 25
Monitor, high resolution, 214
Motion blurring, 5
Mouse, 100, 101, 103
Mouse icon, 42
my_data, 55

Negative numbers, 219
Newton's Laws of Motion, 29, 137
Nibble, 218
Noncommutative products, 84
Number systems, 218

Object frame, 27, 28, 87
Object reference frame, 85
Object-to-world transform, 29, 85, 87
Observer, 29, 81
Operating System, 101
Operating System, calls to, 221
Order of Rotation, 84
ORG, 32
otranw, 87, 90

Packet handler, 102
Painter's algorithm, 110, 178
Patch, 186, 186
PC (program counter) relative addressing, 7, 210
Personal Iris, 5
perspect.s program, 71, 74
Perspective transform, 29, 65, 69, 72
Physical realism, 137
Physical screen, 10, 51, 87
Pilot, 152
Pitch, 182
Pixar, 5
Pixel colour, 13
Pixels, 37
Plane, hither, 162
Plane, Yon, 162
polydraw, 73
polyfil0.s program, 16, 21
polyfil1.s program, 15, 16, 18
polyfil2.s program, 41, 44
Polygon Fill, 40
Polygon Mesh, 3
Polygons, 52
Polyhedral structures, 25, 26, 33
Position dependency, 7
Principal Axes, rotations about, 82
Products of Vectors, 239
Program Counter Relative addressing, 210
Projection, centre, 66
PTSIN, 14
put_pixl.s program, 14, 17

Radiosity, 115
RAM, 209
ramview.s program, 101, 104, 105
Raster scan graphics, 25, 36
Real-time, 26
Records, 178, 179
Reflection, 129
Register Direct/Indirect addressing, 210
Registers, 209
Relocatable code, 7, 8
Resolution, 11
Reyes system, 5
Right-handed Cartesian coordinates, 30
Roll, 182, 183
Rotation, 129, 130
Rotation, about
    arbitrary axis, 152
    principal axes, 82
    x-axis, 83
    y-axis, 83
    z-axis, 83

Rotations, composite, 84, 129, 130
Rotations, order of, 83
Rotations, simple, 81, 129
Row vectors, 32
Running times, 184

Scalar (Dot) Product, 112, 239
Scaling, 129, 133
Scan conversion, 25
Screen, 10, 29, 51
Screen buffering, 10, 87, 89
Screen coordinate system, 30
Screen frame, 67
Screen pixel, 11
Screen RAM, 30, 51
Screens1 and 2, 87
screenflag., 88
Self-similar structure, 4
SETSCREEN, 104
SETSCREEN (#5), 101
set_pixl.s program, 15, 24
Shear distortion, 133
Simulators, 4
Sines, tables of, 88
Sorting, 178
Sprite graphics, 115
SPRITES, 26
Square root, 114, 119, 164
ST cube, 136
ST monolith, 136
ST operating system,.221
Stack pointers, 209
Standard palette, 13
Status register, 210
Surface Normal Unit Vector, 111, 113, 241
Sutherland-Hodgman Algorithm, 51 - 55
System Variables, 101
systm_00.s file, 15, 23
systm_01.s file, 41, 43, 50
systm_02.s file, 89, 97
systm_03.s file, 104, 106
systm_05.s file, 187, 205

Tables, 88
Tile, 178
Title, 177
TOS, 9
Transformations, 27
Transforms, 31, 243
    general in 3D, 129
    geometric, 129
TRAP, 15
Trigonometric tables, 88
trnsfrm.s program, 137, 140

UNLK, 73

Variables, 32
vblank, 87
VDI, 8, 221
Vectors, 25, 31, 32, 111, 236
Vector (Cross) Product, 113, 160, 239, 240
Vector graphics, 25, 26
Vector table, 102
Vertical blank, 10, 12, 87
View frame, 29, 66, 67, 81
View frame base vectors, 156, 182
View plane, 66
View port, 30
View vector, 112, 114
Viewing transform, 180
Viewpoint, 66, 112
Visibility, 111, 113
Visibility sort, 186
VT52 Terminal emulator, 104
VT52 Terminal Escape Codes, 224

Window, 66, 88
Windowing, 30, 51, 55
Wire frame, 25
Words, 210, 218
World frame, 28, 81, 85, 156
World map, 186
World picture, 81
World Scene, 176
wrld_scn.s program, 182, 184, 185, 189
wrld_vw.s program, 163, 165
wrt_phys_tbl, 15

X-axis, rotation, 83
XBDOS, 221
XBIOS, 8, 9, 15, 100, 221
XBIOS call number $25, 89
XBIOS call number 6, 119
XBIOS calls, 223
xbuf, 16, 42, 51

Y-axis, rotation, 83
Yaw (bearing) pitch, 182, 183
Yon plane, 162

Z-axis, rotation, 83

Interested in high performance graphics programs? Here's how to write them – in full colour, 3-D and real-time!

This accurate and up-to-date book shows how to produce vivid 3D solid graphics in colour, in real-time. Not just pretty pictures that take half an hour to draw, but everything you need to write a flight simulator. The book is aimed at the ambitious Atari ST user, but users of other systems will see how to implement the techniques. The contents include: assembler programming; screen mapping; colour palettes; real-time solid graphics; raster line drawing; solid 3-D; transformations – rotations, translations, enlargements, shears; input from the joystick, mouse and keyboard; hidden surface removal and illumination from a light source; building a 3-D world of animated objects and flying around it under joystick control.
EACH SECTION COMPLETE WITH WORKING EXAMPLE PROGRAMS – *READY TO RUN!* PLUS – *ALL PROGRAMS AVAILABLE ON DISK!*
No less than *EIGHT TECHNICAL APPENDICES* complete this definitive book.
Truly **the ultimate ST reference book** for the serious user.

Andrew Tyler is a research physicist at Manchester University. This fabulous new book has been developed from Andrew's popular series of lectures on assembly language programming. His students found that the subject took on a new dimension – quite literally – when they applied assembler to the challenge of real-time computer graphics.
Now, all ST users can be at the leading edge of this exciting area!

About Sigma Press:
We publish a wide range of books on all aspects of computing. Write or phone for a complete catalogue:

Sigma Press, 1 South Oak Lane,
Wilmslow, Cheshire SK9 6AR

Phone: 0625 - 531035

We welcome new authors.

ISBN 1-85058-217-3

9 781850 582175

SIGMA
PRESS